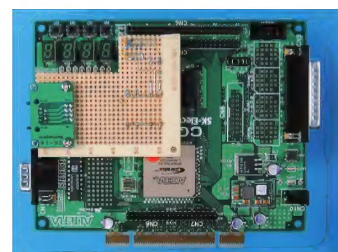
Interface

2004 4

Contents



［表紙デザイン：㈱プランニング・ロケッツ］





特集

Ethernetを基本から理解するために

作りながら学ぶ Ethernet活用技法

Cover Story
Application techniques of Ethernet through making 35

Interface

2004 4

Contents

話題のテクノロジ解説

ショウレポート&コラム

一般解説&連載

情報のページ

連載「TOPPERSで学ぶRTOS技術」,「フリーソフトウェア徹底活用講座」は,お休みさせていただきます.

▶ エレクトロニクスの総合展示会

第21回 エレクトロテスト・ジャパン

北村 俊之

プリント基板のテストや検査を行うための技術や装置に関する展示会「第21回エレクトロテスト・ジャパン」が1月28日(水)～30日(金)の3日間，東京ビッグサイトで開催された．主催はリードエグジビション ジャパン(株)．同展示会は，1972年にエレクトロニクス製造技術展としてスタートし，世界8か国10か所で開催されている．同じ会場には「第33回インターネプコン・ジャパン」を中心に，「第5回半導体パッケージング技術展」，「第5回プリント配線板EXPO」，「第5回電子コンポーネントEXPO」，加えて光通信システムやデバイスを一堂に展示する「第4回ファイバーオプティクスEXPO」も併催され，エレクトロニクスについての一連の技術がチェックできる催しとなっていた．今回は，「エレクトロテスト・ジャパン」および「ファイバーオプティクスEXPO」にスポットをあててレポートする．

● 第21回エレクトロテスト・ジャパン

テスミックでは，画像処理による検査装置中心の展示を行っていた．異形部品画像検査装置「V-CATS」(写真1)は，4台の固定カメラを利用して，ボードなどの検査を行う装置．カラー画像を扱えるため，ボード上の部品位置や色などが鮮明にモニタに写し出される．これまでは，モノクロ画像しか扱えないものが多かったために，部品の位置や形状は識別できるが，色のチェックまで行うことは難しかった．また，カメラを固定させ，ボードを移動させて撮影を行うために，より安定性のある映像を提供できる点も特徴となっている．

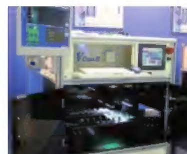


写真1 テスミックの異形部品画像検査装置 V-CATS」

エイト産業では，ディジタル・カメラ・モジュールのための評価システム「NT803DAV」や，簡易評価装置「XGA-BOX」(写真2)など，カメラ付き携帯，小型カメラ専用の量産向け評価，検査装置の展示やデモを行っていた．「XGA-BOX」は，これまで目視に頼っていた小型カメラ出荷時のデフォルト焦点距離などの調整を，画面上から簡単に行うことができる．



写真2 エイト産業の XGA-BOX」

日本ナショナルインスツルメンツでは，信号の集録や解析，データの表示などをグラフィカルに行う「LabVIEW 7 Express」を中心に，「GPIB」，「PXI」など計測ソフトウェアとハードウェア・モジュールを組み合わせたPCベースのテスト環境の展示やデモを行っていた(写真3)．

マイクロ・スクェアでは，目視外観検査用BGA，QFPビデオ・マイクロスコープ「MS-3000 system」のデモを行っていた．同製品は，ハンディ・タイプの小型カメラで，BGAと実装基板のはんだ接合部分をモニタに写し出すことで，目視検査を可能にしている．従来のX線による検査と併用することで，より信頼性の高い検査および解析を実現できるとのことだった(写真4)．



写真3 日本ナショナルインスツルメンツのFPGA評価システム

グラフィンでは，高速，大容量入出力ボード「GPIO」シリーズを中心とした展示を行っていた．「GPIO-3000」シリーズを搭載することにより，447Gバイトの大容量ロガーやジェネレータの構築が可能になる．また，「GPIO-4000」シリーズでは，64ビット×66MHzの高速ロガーやジェネレータの構築が可能になるとのことである．新製品としてA-Dモジュール入力コネクタを搭載したユニバーサル・インターフェース・ボード「GPIO-FPGA」の展示も行われていた．こちらはアナログ･デバイセズのA-D変換モジュール入力に対応し，入出力信号レベル変換機能やUSB2.0インターフェースなどを搭載している．



写真4 マイクロ・スクェアの展示

● 第4回ファイバーオプティクスEXPO

日立製作所では，新製品のギガビットEthernet PONシステム「AMN1500」シリーズ，VDSLとメディア・コンバーター体型の光ハイブリット・システム「AMN1400」シリーズ，IPv6によるマルチキャスト通信を可能にするアクセス・サーバ「AG8100」や光波長多重システム「AMN601GF」を用いたライブ・デモを行っていた(写真5)．「AMN1500」シリーズは，IEEE802.3ahに準拠した双方向最大1Gビットの Ethernet光アクセス・システムとなっており，キャリアからサービス・プロバイダ，ホームまで幅広い環境でギガビット･アクセスを経済的に実現するという．



写真5 日立製作所の光波長多重システム「AMN601GF」

サイバネットシステムは，カナダのオプティウェア社の光導波路解析シミュレーション・プログラム「OptiBPM」，時間領域光伝搬ソルバー「OptiFDTD」や米国のORAの光学設計評価プログラム「CODE V」など光通信システムにおける各種コンポーネントおよびシステム設計のためのソフトウェアの展示を行っていた．

アドバンテスト(写真6)は，光スペクトラム・アナライザ「Q8341」，光パワー・メータ「Q8230」などDWDM，ハイ・ビット･レート通信，光海底ケーブル通信などに使用される光デバイスや光サブシステムを試験，評価するための計測器の展示を行っていた．「Q8341」は，0.5sのスループット測定，0.001mm分解能のコヒーレンス測定などを特徴としており，波長範囲は350nm～1000nmに対応している．



写真6 アドバンテストのブース

アンリツは，ネットワーク・パフォーマンス・テスタ，光サンプリング・オシロスコープ・システム，IPネットワーク・アナライザなどの展示を行っていた．ネットワーク・パフォーマンス・テスタの新製品である「MP1590A」(写真7)は，PDH，DSn，SDH/SONET，OTN装置の試験やジッタ測定を可能にする測定器である．ランダム・エラー挿入機能や光出力可変機能を装備しているため，複数の測定器を必要とせず，単体でFEC機能を評価できることを特徴としているとのことだった．ジッタ測定機能や外部光入力機能は，プラグイン・ユニットとなっており，用途に応じてユニットを組み合わせることができる．

アジレント・テクノロジーでは，ディジタル・コミュニケーション･アナライザなどが一体となった「Infiniium DCA広帯域オシロスコープ」や光ファイバ通信用850nmマルチ・モード対応機器などの展示やデモを行っていた．



写真7 アンリツのネットワーク・パフォーマンス・テスタの MP1590A」

Show & News Digest

日本TI,1GHzで動作するDSPを出荷開始

■ 日時：2004年1月20日(火)
■ 場所：日本テキサス・インスツルメンツ(東京都新宿区)

日本テキサス・インスツルメンツ(株)は，90nmプロセスを採用し1GHzでの動作を実現したDSP TMS320C6414T/15T/16Tを発売した．同社のDSPはローエンドからハイエンド製品まで3種類のラインナップが存在するが，今回発表されたDSPはハイエンド製品のC6000シリーズである．C6414Tは，1GHzの動作周波数で8000MIPS，4000MMACs(百万積和演算/秒)の性能をもち，1Mバイト以上の内部メモリ，64チャネルのEDMAを搭載している．

また，従来製品である720MHz動作のC64xデバイスも90nmプロセスを導入することにより，低価格での販売を開始するとのことだ．



日本TI代表取締役 K.Bala氏



DSP C6414T/15T/16T

クライアントのバックアップが可能なDR-Cube

■ 日時：2004年1月27日(火)
■ 場所：報映産業(東京都中央区)

報映産業(株)は，バックアップ・アプライアンス・サーバ「DR-Cube」を発売した．従来の企業向けバックアップ製品はサーバを対象にしたものが多かったが，DR-Cubeはクライアントのバックアップを主眼におき，サーバ機器+サーバ・ソフトウェア+クライアント・ソフトウェアをセットにした製品．クライアントに定期バックアップを行うソフトウェアを常駐させ，ネットワーク上に設置したサーバへ定期的にバックアップを行うというもの．「企業のデータの60%がサーバでなく，クライアントのPCに置かれている」(同社取締役社長 朝日 宏章氏)という報告もあり，企業でのクライアント・データのバックアップに対する需要が高まっていることから，この製品を発売することになった．

サーバ用OSとしてはWindows Storage Server 2003，データ・バックアップ・ソフトウェアとしてはDantz Retrospectを搭載している．対象クライアントはWindows 95以降(NT 4.0を含む)，MacOS 7.1以降，Red Hat Linux．価格はオープン・プライス．5クライアントまでのライセンスが付属し，クライアント数を無制限にするオプションもある．



DR-Cube

アナログ・デバイセズ，750MHzのデュアル・コアDSP「ADSP-BF561」を発売

■ 日時：2004年1月27日(火)
■ 場所：大手町サンケイプラザ(東京都千代田区)

アナログ・デバイセズ(株)は，750MHzのDSPコア「Blackfin」を二つ備えたプロセッサ「ADSP-BF561」を発売した．処理性能は3000MMACs．ポータブル・メディア・プレーヤなどのディジタル民生機器での利用を想定している．

本プロセッサは，二つのDSPコアと共有メモリを備えている対称型マルチ・プロセッシング(SMP)アーキテクチャとなっている．たとえば，偶数番のフレームを一方のコアが，奇数番のコアをもう一方のコアが処理するといったことができる．

現在，サンプル出荷を行っている．量産出荷の開始時期は2004年第2四半期．1万個購入時の単価は$39.95である．

また，600MHz品と，シングル・コアの製品(750MHz品と600MHz品)も用意している．



ADSP-BF561

NTTソフトウェア，セキュリティ機能を強化したWeb言語「CipherCraft/Curl」を発売

■ 日時：2004年1月20日(火)
■ 場所：NTTソフトウェア(東京都品川区)

NTTソフトウェア(株)は，Web向けプログラミング言語Curlにセキュリティ機能を付加した「CipherCraft/Curl」を発売した．CurlはWebベースのアプリケーションを作成できるプログラミング言語だが，3Dグラフィクス・アプリケーションを実現できるなど，ユーザにリッチなGUIを提供できることが特徴である．また，統合開発環境Surge Labも提供される．

今回国内で提供するCurlは，Curl Client/Web Platform V3.0をベースにしている．NTTソフトウェアは官公庁・地方自治体向けにPKIソリューションを提供してきた．この実績を活かし，「CipherCraft/Curl」はCurlに純国産暗号アルゴリズムCamelliaを使った暗号化機能を付加することにより，安全なWebアプリケーションが作成できるとしている．

ハッカーの常識的見聞録



今月の常識

サーバも仮想化でトラブルに対処しよう

広畑 由紀夫

☆ 今回はVirtual PCを使用して，簡単に現在のシステムを移行する手順，および今後のサーバ上のハードウェア・トラブル対策を考えてみます．

ここ数年間の仮想化技術の進歩には目を見張ります．単にOSの内部構造そのものの変更にとどまらず，物理的・論理的に仮想化を施すことによって得られる利点が注目され，また技術の進歩にともなって仮想化技術そのものが安定してきたことにより，仮想化が盛んに行われているのでしょう．

● Virtual PC 2004

マイクロソフト社がConnectix社より買収したConnectix Virtual PCシリーズの，マイクロソフト社による初の製品およびサポートとなる製品です．昨年のカンファレンスにおいて，Longhornプレビュー版とともに参考配布されました．国内において，日本語版の製品出荷は1月末の段階ではまだ行われていませんが，英語版はMSDN会員向けに配布されているので，すでに入手している読者も多いことでしょう．英語版Virtual PC 2004であっても，日本語OSをライセンスとともに導入することで，今後リリースされるVirtual Server製品への準備，および導入テストは可能だと思われます．

● V2I Protector

ハードディスクを物理セクタ単位でバックアップすることの可能なツールの一つで，Drive Imageシリーズで有名なPower Quest社の製品です．CDからのブートによって，ネットワーク上のイメージ・ファイルからのバックアップにも対応するため，すでに導入している読者も多いかもしれません．今回は，このツールを使用して物理ドライブを仮想化してみます．

● 物理ハードディスクを仮想ドライブに移行する

まず，V2I Protectorを使用して，物理ドライブのデータを可能な限りすべて抜き出します．このとき，ハードディスクがクラッシュしていることも想定して，エラー・チェック，およびエラー・コンティニュを行うフラグを設定して，650Mバイトくらいの分割ファイルか，CD-Rへの書き込みバックアップを行います．したがって，このとき物理的にクラッシュしているセクタが存在している場合でも，可能な限り対処できるはずです．次に，Virtual PCを起動し，新規OSの仮想イメージを作成します．その際，実際に復元するOSを選択することを忘れないようにしてください．たとえば，Windows 2000 Serverを仮想化するのであれば，Windows 2000の仮想イメージを作成するということです．これは，復元およびインストール完了後にVirtual PC 2004の拡張機能をOSに組み込むときに参照されるようです．

仮想化する準備ができたら，次はいよいよ復元です．CD-Rにバックアップを取ったほうは，物理ドライブにCDをマウントしてCDブートにより復元してもよいでしょう．しかし，筆者が行ってみた感じでは，いったんISOファイルにCD関連のツールを使用してイメージにし，それをVirtual PC 2004の機能でドライブとしてマウントするほうがよいようです．

● IP割り当て

復元が終わったら，IPアドレスなどを割り当てます．Virtual PC 2004では，複数のNICを共有して割り当てることが可能です．そのため，2枚のNICで内部・外部と分けているサーバであっても，ホストPC(Virtual PCを動作させるPC)が2枚のNICを搭載していることで，そのままの環境に似せることができます．さらに，NIC自体が仮想化されているため，複数台の仮想サーバを同時に起動したとしても，NICを増やすことなく，独立して設定，使用できます．

● 実際のテスト

実際にはホストPCの起動時に，仮想化されたサーバも起動しなければテストになりませんが，現在のバージョンではアプリケーションなので，起動だけでは実行してくれません．

そこで，Windows XPをホストOSにする場合は，自動ログオンするようにし(もちろんセキュリティを施したうえで)，仮想イメージ・ファイルのショート・カットをスタート・アップに登録することで，起動から仮想OSの実行までを自動化できます．

● さらなる仮想化

このように，現在のVirtual PC 2004でも，アプリケーション・レベルでありながら，サーバ内のハードディスクを仮想OSへ移行してテストすることが可能です．今後のラインナップとして，Virtual Serverが予定されているそうです．さらに今後のプロセッサの高速化を併せて考えると，仮想コンピュータそのもののサービスがより身近なものになってくると考えられます．

仮想化してファイルとして扱えるOSにすることで，バックアップやメンテナンスの複雑さから解放され，さらにスケール・アップやスケール・ダウンなどはホストPCの性能を変更するだけでよいという容易さが確保できると筆者は考えています．ハードディスク・クラッシュによるサーバ停止という最悪の事態が起こる前に，コストを抑えて対処しやすくしておくことも今後の課題ではないかと思います．

ひろはた・ゆきお　OpenLab.

IPパケットの隙間から

いまだに結論が出ていない UUCP接続の話

祐安 重夫

先月号は年末進行で締切が早かったので，前回紹介した事件が始まってからすでに2か月以上が経過した．先月の原稿を書いた時点では，今回の原稿を書くまでにはaからZまで一巡して，すでに事態は収まっているのではないかと楽観していたのだが，それはとんでもなく甘い予測だった．

一巡し終わる前に次のサイクルが始まり，一度にやってくるメールの本数は，増加の一途をたどった．1回の接続にかかる時間が1分を超えたあたりで，KDDIの使用を断念して接続をNTTに切り替えたが，週に3回の接続で1回に5分以上かかるようになったところで，UUCP（Unix-to-UNIX Copy）接続を全面的に閉鎖した．

すでにエラー・メールは送信しないように手を打ったのだが，UUCP以外の接続は従量制ではないので，大量のSPAMを受信するのにかかる経費がバカにならないのである．

問題を整理してみよう．SPAMの送信がSMTPサイトに対して行われる場合は，SMTPサーバが受信人の有無をその場でチェックし，存在しないユーザへの送信を拒否するので，SPAMの送信側にも相手が存在しないことはすぐにわかる．SPAMメールの送信元は偽造されているのが普通だが，この場合は送信元にもその情報が伝わる．

UUCP接続の場合，最初に受信するメール・サーバはSMTPだが，実際のUUCP受信元にどのようなメール・アドレスが存在するかはわからないので，無条件にメールを受け入れ，UUCPパケットに変換して送り出す．この場合，SPAM送信元には，そのメール・アドレスが存在するようにしか見えない．存在しないアドレスには二度とメールを送らないように作られた「よくできた」SPAMプログラムなら，ここであきらめてくれる．

だが，UUCPの場合，最終的に受信してみないと，そのアドレスが存在するかどうかわからないので，受信人がいない場合（今回の事件ではほぼ100%が存在しない）偽造された発信元への大量のエラー・メールが送信される．その偽造された発信元も，今回のケースの場合ほとんどが存在しないアドレスだったので，それに対するエラー・メールがまた大量に戻ってくる．

UUCPをメールの配送に使っているサイトはあまりないだろうし，うちの場合もメール以外に必要はないし，そのメールもpostmaster以外は一つだけしか有効ではない（さらに実際にそのアドレスにメールが来ることは，現実にはほとんどない）．UUCPなのは，とりあえず使い続けているし，それまではいちばん安価だったという理由にすぎない（それが今回の事態で，一気に高価なものになってしまったのだが）．

UUCPが現在使われている現場は，おもにネット・ニュースの配送であり，当然だがUUCP over IPが主流である．実際にネット・ニュースに限ってみれば，NNTPと違ってUUCPはgzipによって圧縮されたニュースの本文を配送するので，効率が良いのだ．もちろん，それを利用しているサイトのほとんどが，メールを含むそのほかの接続にはIPを利用しているのはいうまでもない．

ところで，UUCP以外にこのような被害にあうサイトは，存在しないのだろうか．実際には外から見えるSMTPサーバがHUBホストになっていて，実際にメールを受信する子SMTPサーバに配送するようなしくみは，ある程度大きなサイトでは行われている場合があり，こういうところも被害にあっているだろう．だが，そういうところは費用も回線容量も十分にあり，全体のメール・トラフィックに比較すれば，相対的に大した被害にはなっていないし，実際の被害にはあまり気がついていないのかもしれない．

しかし，うちには，義太夫も浄瑠璃も鈴木大拙も松尾芭蕉も市川団十郎も玄奘三蔵も弁慶も国木田独歩も，ましてや名無しの権兵衛もだれもいない．本当にこんなアドレスにメールを送っていて，どんな効果があるのだろうか（もちろん，それらしいアドレスも含まれているのだが）．

ところで，メールの内容だが，最初のうちは特定のURLへのリンクを持ったHTMLメールだったのだが，最近では解読できないBASE64ファイルのことが多かった．だが，ウィルスではなかったようだ．

ウィルスといえばまた新たなものがはやっているようだが，筆者のところではWindowsでメールを受信するなどということはしていないので，感染についてはまず心配ないし，ウィルス・メール自体もそれほどの頻度ではやってきていない．ただ，不特定多数からの問い合わせメールを受け取るようにWeb上で公開されていたり，メール・ニュースの配送元になっているアドレスにはウィルスがけっこう届いているようだ．同時に，ウィルスが送信先不明で届かなかったというエラー・メールが，その半分くらい届いていることも，かなり迷惑な話である．ウィルスに感染したコンピュータに入っているメール・アドレスは，送信先にも差出人にも流用されるということである．

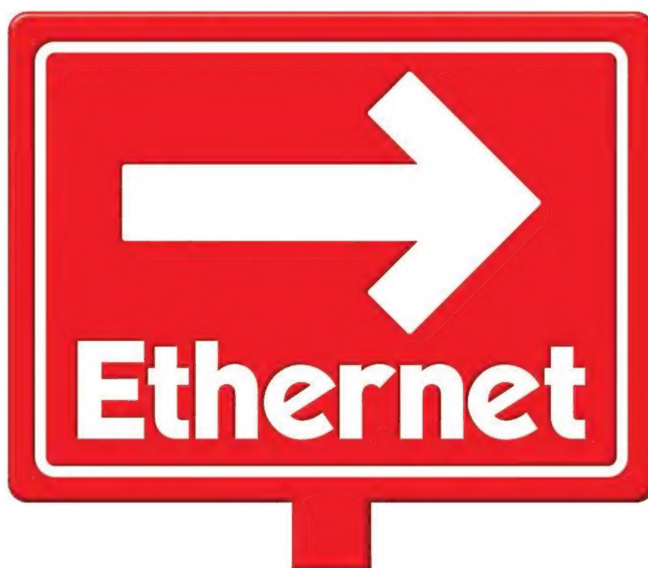
今回のUUCPによるSPAMでも，差出人偽造とその差出人がその時点では実際には存在しないことが，問題を大きくしている．

ところで，最終的には，UUCP接続をやめるべきなのだろうか．

すけやす・しげお　インターメディアアクセス

特集 Ethernetを基本から理解するために

作りながら学ぶ Ethernet活用技法



地上ディジタル放送対応機器やDVDビデオ・レコーダなど，ネットワークに接続できる情報家電が次々登場している．組み込み機器へのネットワーク機能の必要性は日に日に高まっている．

たとえば，市販のLANコントローラとRTOSやTCP/IPプロトコル・スタックを組み合わせてシステムを構築したとしよう．不具合なく順調に動けば問題ないが，何か一つでも不具合が発生した場合に，あなたは対処できるだろうか．あまりに高機能なLANコントローラは，膨大な数のレジスタをもち，使い慣れないOSやプロトコル・スタックでは，単純なパケット送受信テスト・プログラムを書くのもたいへんである．ブラック・ボックスの組み合わせでは，どこに不具合の原因があるのかすらわからない場合も出てくる．やはり物理的なレベルから，Ethernetの動きを理解しておく必要があるだろう．

そこで今月号ではLANコントローラをHDLで設計し，実際にFPGAに実装してみることで，Ethernetの基礎を物理レベルから理解することを目ざす．FPGA以外に使用する部品は，パルス・トランスと抵抗，コンデンサ，そしてRJ-45のコネクタのみで，可能な限りブラック・ボックスとなるような部品は使わずに，LANコントローラを実現する．

また，各種テスト・プログラムを使って，製作したLANカードが本当に既存のネットワーク機器と通信できるかどうかを確認する．さらに，Linux用のデバイス・ドライバを作成して，Linux上から実際にLANカードを使ってみる．

特集の最後では，オープン・ソースの組み込み機器向けTCP/IPプロトコル・スタックについての解説も行う．

プロローグ



Ethernetのハードウェアを理解しよう!

松本 信幸

● Ethernetあれこれ

ローカル・エリア・ネットワーク技術としてEthernetが登場して幾星霜．初期においては，当時のEthernetである，10Base-5とともに，TokenRingやFDDIといった技術も利用されていました．少し後に登場した10Base-2は，10Base-5より多少操作性がよく，そして安価なため，Thin-Ethernet(細いイーサネット)やCheeperNet(イーサネットの語呂に似せて，安価を意味するCheepと掛け合わせたことば)と呼ばれました．つまり当時はローカル・エリア・ネットワークにおいてEthernetといえば10Base-5を指していたのです．一時的とはいえ，同じようなメカニズムで動作するものでさえ，Ethernetとは違う名称が付けられていました．

しかし時は流れ，TokenRingやFDDIもお目にかからなくなりました．それでもEthernetは，インターネットを支える技術として依然存在し続けています．とはいえ，現在耳にするEthernetは10Base-5を指しているのではありません．またTokenRingやFDDIが発展しなかったわけではありません．TokenRingも初期の8Mbpsから16Mbpsへと速度向上を図りました．FDDIに関してもケーブルの作業性を向上させるために同軸ケーブルを用いるCDDIやツイスト・ペア・ケーブルを使用するTPDDIというものも検討されましたし，速度を向上させ，新しいアプリケーションにも対応できるようなFDDI-IIもありました．

新しいアプリケーションに対応し，高速のインターフェースを提供できる技術として，ATM-LANが話題になったときもありました．これは電話網の流れをくむ局用の通信技術として登場したATMをローカル・エリアの世界に持ち込もうとしたものです．これはEthernetのライバルとしての技術ではなく，Ethernetを取り込み，共存する技術として一時期発展しました．ATM技術の初期においては音声のような連続情報を格納するAAL Type1とパケット形状の情報に用いるAAL Type3/4の二種類しかなかった(AAL Type2は欠番：後に新しいフォーマットで再登場)のですが，Ethernetフレームとの整合性をならびにTCP/IPの動作との整合性を考慮してAAL Type5が新設されたという経緯もあります．

しかし，ATMも本命とはなり得ませんでした．個人的には，ATM Cellの大きさを53バイトという素数を用いるという愚行を行ったことが原因の一つであると思っています．64バイトの長さをカウントするためには8ビット・カウンタを2段組み合わせればカウントすることは可能ですが，素数の場合にはそうはいきません．このほかにもいくつかの要因があり，エッジ系に配置する機器のアップリンク・インターフェースを安価に用意することができなかったのです．

ATMを尻目に，Ethernetは独自の進化を遂げます．インターフェースの速度は10倍ずつ増加していきます．にもかかわらず，UTPを用いるインターフェースにおいてはオート・ネゴシエーションを用いた自動切り替えによりインターフェース速度を決定し，VLAN-tagの利用やMPLSにより，いろいろなアプリケーションにも対応可能になりました．さらにUTPケーブルに便乗させる形で電力を供給する技術も確立されるに至り，バリエーション豊富な技術として，不動の地位を築いたといっても過言ではないでしょう．

● Ethernet ≒ CSMA/CD!?

昨今は，PCショップをのぞいてみると，10M/100M両対応の機器が一般的で，5ポート程度のハブが2,000円も出せばお釣りがくるような時代です．さらにギガビットにも対応した機器も，手ごろな価格で次々と登場しています．

このようなときに，なぜ今ごろ10Base-Tインターフェースを手作りしようなどと試みたか…それは，Ethernetの基本はCSMA/CD(Carrier Sense Multiple Access with Collision Detection)方式にあると考えているからです．もしここで，NE2000互換のコントローラなど市販のネットワーク・インターフェース・コントローラ(NIC)を使ってしまうと，衝突検出および再送処理はコントローラが自動的に行ってしまうので，それを体験的に学習したり試してみることができません．そこであえて，NICそのものを作ってみることにしたのです．

また，現在のEthernetを支えるコア技術は一つだけではなく，たくさんの技術が組み合わされて実現されています．この技術の中にはローカル・エリア・ネットワーク技術の流れを組まないものも入っていますし，新しいものばかりというわけでもありません．一例をあげると1000Base-Tのような新しいインターフェースであっても，実現するために新たに導入された技術は，新しいものばかりではなく，古くから別のシステムで使用されていたものを流用しているものも少なくありません(たとえば1000Mbpsという高速で全二重通信を実現するために使われた技術は，もともと固定電話で培われた技術を流用している)．

つまり，将来，10Base-Tを支え発展させた技術が，概念としてまったく異なったシステムに流用されることがあっても良いのではないかと思っているのです．ネットワークが宅内にど

んどん入り込んできていますが，今後登場してくるものの中にはCSMA/CDを応用した，Ethernetではないネットワークや装置があるかもしれません．

たとえばインターネット電話といった新しいシステムに対応するためには，CSMA/CD方式を用いず，全二重による通信を行ったほうが良いことはわかっています．しかし「基礎なくして応用なし」です．規格の流れからすれば，高速化のために消えつつあるCSMA/CD方式ではあるのですが，このCSMA/CD方式による通信をもう一度しっかり見直し，理解してこそ，これから登場してくる新しい技術を，より深く理解できると信じています．

● 特集の構成について

今回の特集の章立ては次のようになります．

第1章では,(お約束どおりだが)Ethernetという技術の概要について解説します．

第2章では，そのEthernetの中から，今回特にスポットを当てて開発を行う10Base-Tについて，その特徴を見ていくことにします．

第3章では，10Base-Tの動作を確認しつつ，オリジナル仕様のEthernetコントローラの設計を行います．今回の開発は編集部の意向もあり，FPGAをベースとしたものとなります．外付けする部品は，パルス・トランスと数個のコンデンサと抵抗のみという非常に簡単な回路です．またフリー版のFPGA開発ツールで論理合成や配置配線も可能で，HDLソースも公開するので，だれもが実際に試せるようになっています．

第4章では，実際に製作したEthernetコントローラについて，正常に動作しているかどうかの確認を行います．今回はFPGAをほぼ直結しただけなので，Ethernetの仕様を100%満たせているわけではありません．また一言で10Base-Tといっても，実際に製品として出回っているハブやLANカードには，細かな点でいろいろと動作の異なるものがたくさんあります．ここではオシロスコープやロジアナなどで，実際の波形を観測したり，いろいろなハブやLANカードと接続して，実際にパケットのやりとりができるかどうかをテストします．

第5章では，今回製作したEthernetカードに対応したLinux用デバイス・ドライバを作成し，実際に製作したEthernetカードをLinux上から使ってみます．Linux側でSambaを立ち上げ，Windowsマシンから見えるネットワーク・ドライブにファイルをコピーするなど，手作りのEthernetカードで実際に本格的なネットワーク・アプリケーションが動いているところを見ると，やはり感動します．

第6章では，組み込み機器にEthernetを採用する場合に必要となるTCP/IPプロトコル・スタックについて解説します．ここではITRON準拠のTOPPERS/JSPと組み合わせて使えるTINETについて解説しています．ターゲットとなるプラットホームはH8マイコン・ボードにRTL8019ASという非常にポピュラなものなので，各分野に応用が利くと思われます．



写真　今回設計したオリジナル仕様のLANカードを実際に動作させているようす

左のPCに今回設計したLANカードを実装し，Linuxを起動してSambaやhttpサーバを走らせている．右のPCはWindows 2000を起動し，ネットワーク・ドライブでファイル・アクセスを行い，WWWブラウザでテストWebページを表示させているようす．

右に重ねたハブや，左に並べた市販されている各種LANカードのすべてと問題なく通信できることを確認している．

第7章では，Ethernetを搭載したCQ RISC評価キットの活用事例として，組み込み機器向けに最適化したTCP/IPプロトコル・スタックを開発し，転送性能などを評価しています．こちらもスタックはオープン・ソースとする予定とのことで，近日中には公開されると思います．

さらに今回はAppendixとして，ネットワーク装置を開発する際に関連してきそうなEthernet技術の中から，Power over Ethernetについて概要の紹介もしています．

● 今後の展開

今回の特集は，もっとも基本的な動作を理解することが目的であるため，10Base-Tの半二重動作のみとなっています．

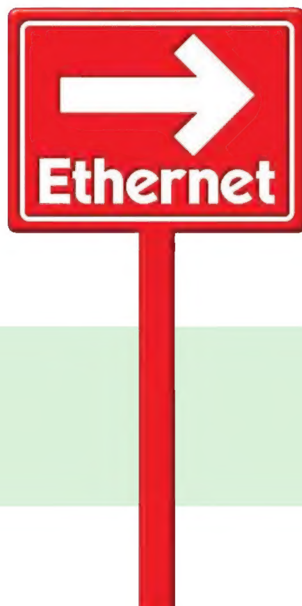
しかし近い将来，今回の設計をベースにいくつかの機能を加えていきたいとも考えています．まず次のステップとしては，オート・ネゴシエーション機能を加えて，10Base-Tの全二重でも動作できるようにしたいと考えています．

そして次に，100Mbpsの速度に対応した100Base-TX(Fast Ethernet)への対応を実現させ，オート・ネゴシエーションと組み合わせることで，100Base-TXの全二重と半二重，そして10Base-Tの全二重と半二重で動作可能なものにしていきたいと思っています．

とりあえずは，基礎の基礎である10Base-Tについて解説していきます．

まつもと・のぶゆき　(株)タムラ製作所

第1章



Ethernetの種類と動作原理を理解する

Ethernetの基礎知識

松本 信幸

Ethernetの動作原理の基本は，CSMA/CDと呼ばれる衝突検出方式にある．また「10Base-T」という表記にも意味や命名規則がある．第3章以降でFPGAでネットワーク・コントローラを設計する際にも，パケット・フォーマットなどの知識は必須である．ここではEthernetの特徴やEthernetフレームの構造，Ethernetの種類などについて解説する．
（編集部）

1 Ethernetというもの

ことばのもつ意味は，時代とともに変化していきます．あるものは，本来とはまったく異なった意味のものへと変化し，またあるものは本来の意味合いを含みつつ，より広範囲の解釈ができるものへと変化します．Ethernetということばは後者にあたります．現在，Ethernetと呼ばれているものは，昔の技術を含みつつ，まったく新しい技術にも対応しています．もっとも，その中には消えていった枝葉があることも事実ですが．

最近の10ギガビットEthernetなどを見ていると，「いったいどのあたりがEthernetなのか？」と，表向きには昔の面影などまったく見えません．しかし，よく見ていくと所々に昔ながらのエッセンスが加わっていることを見つけることができます．そして，誕生初期に確立された技術も，依然として利用されているので，ひと言にEthernetといっても，現在では広範囲なものとなっているということになります．

今回は，このEthernetに関して，技術的に基本となる部分を見直す意味で，初期の技術について見ていこうと思います．しかし，昔の技術を包含しつつ発展しているといっても，ほとんど消えてしまった部分も当然あるので，今でも利用されているものの中で古いもの，言い換えれば，現在のEthernetのもっとも基本となるべき10Base-Tにスポットをあてて見ていくことにします（図1）．

なお，最近では10Base-TのIEEE802.3iは単独ではなくIEEE802.3の規格書の13章および14章に入っています．

2 Ethernetの特徴と思想

Ethernetの黎明期に活躍したネットワークは，実際のところ単なる同軸ケーブルでした．ケーブルの被膜が黄色であったことからイエロ・ケーブルとも呼ばれました．この同軸ケーブルの両端に無反射終端（ターミネータと呼ぶ）をつけた，最大長500m（1本のケーブルとしてであって，リピータ接続を行った場合は2500m）の何の変哲もないケーブルがネットワークでした．

この同軸ケーブルに四角いタップ・トランシーバという箱を，

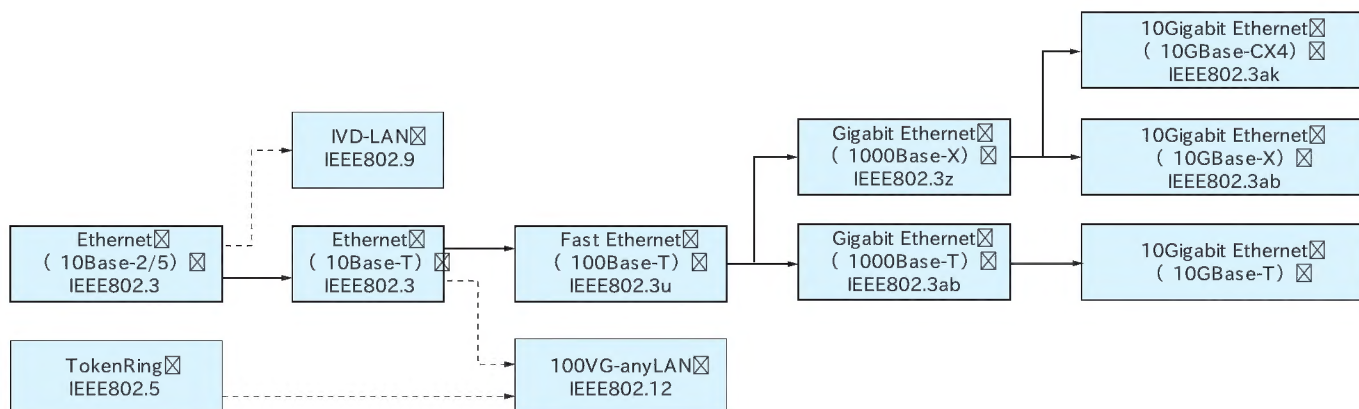


図1　Ethernetの流れ

ケーブルに「噛み付くように」とりつけ，そのタップ・トランシーバと，端末であるコンピュータの間をトランシーバ・ケーブルと呼ばれるケーブルで接続するようになっていました．

Ethernetとして規定されている通信方式のもっとも基本部分に位置する概念は，自己責任による通信と考えることができます．言い換えれば，通信を行う場合においてネットワークの動静を監視しているマスタから，いちいち送受信の許可を得るというようなものではなく，ネットワーク上に存在するそれぞれの端末自身が自立的に送信が可能な状態であるかどうかを確認，判断し，動作を行うということです．この動作を実現するために用いられる手法がCSMA/CD（Carrier Sense Multiple Access with Collision Detection）方式です．ネットワークに接続されたそれぞれの端末が，このCSMA/CD方式による動作を行うことによって，全体としての通信を実現します．

この手法のメリットは，それぞれの端末が自立的な動作を行うがゆえに，端末の追加やネットワーク構成の変更が容易にできるという点があげられます．たとえば，街の電気屋さんで買ってきた電話機にどのような設定を行おうが，ネットワークである電話網に勝手につないでも通信はできません．無論，親子電話のようにすることはできますが，これは端末が2台になったのではなく，マイクとスピーカが2セットになっただけで，同時に異なった通信はできないので増えたとはいえません．当然，2台の電話機を2ワイヤのケーブルで直接接続しても何の役にも立ちません．電話機を使えるようにするためには交換機が必須であり，交換機によって構成されるネットワーク側で設定を行わなければなりません．

しかし，Ethernetの場合では，そのネットワークのルールに従った範囲内で設定を行うだけで，ネットワークに接続し，通信を行うことが可能です（図2）．また，2台のEthernet端末をクロス・ケーブルなどで直接接続して通信を行うことも当然可能です（逆に，同軸ケーブルに何か「設定」してみろといわれても困るが…）．

3 Ethernetの動作原理 ——CSMA/CDとは

ネットワーク上で動作するにあたり，どこかのマスタから許可をもらうのではなく，自己の判断で通信を行うための手法として用いるCSMA/CD方式の前提には，「ネットワーク・リソースは共有されるべきものである」ということがあります．

基本的な動作を大ざっぱに言うと，それぞれの端末はネットワークである同軸ケーブル上に，ほかにデータがなければ送出しようとするデータを置きます．どれかの端末に向けて送出するのではありません．無論宛て先は示されてはいますが，ネットワーク上に相手側の端末が存在するかどうかに関わらず，とりあえずネットワークに送り出すので，ここではネットワーク上に置くと表現しました．

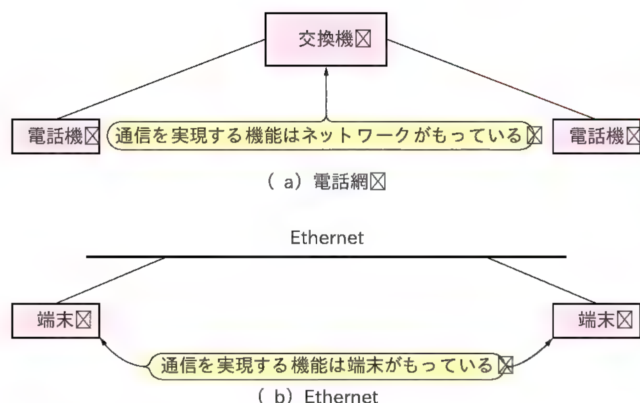


図2　通信を行う機能の所在

そして，データの送出を行っている端末以外のすべての端末は，ネットワーク上にデータが置かれた場合，自分宛てであるものを取り込みます．このため，データとしてのフレームの受信動作は，ネットワーク上に存在するものを，とりあえずすべて受信して，そのうえで宛て先が自分宛てもしくは全員宛てというように，みずからに必要であると思われるものの場合には処理を継続し，それ以外，つまり受け取る必要のないものであればフレームを破棄します．

もう少し詳しく見ていくと，送出動作の大まかな流れは，送出しようとするときに，最初にほかの端末がネットワークに対してデータの送出動作を行っていないということを確認するところから始まります．具体的な動作としては，ネットワーク上に電位差が見えるかどうかで判断します．そもそもEthernetのネットワークの基幹部分は単なる同軸ケーブルなので，そこを基点として電位差は発生しません．どこかの端末が，ネットワークである同軸ケーブルに対して電流を流していない限り電位差の発生はありえないのです．逆に言えば，ネットワーク上に電位差が見受けられるということは，どこか別の端末がフレームの送出動作を行っていると解釈できます（図3）．

ネットワーク上にほかの端末からのデータが存在する場合，それがなくなるまで待ちます．この際，データの送出状況を監視し続けるのではなく，時間を置いて再度確認を行い，データがあるようならば再び時間を置き，なければ送出動作を開始してよいのです．

さて，めでたくネットワークをほかのどの端末も利用していないことが確認できた場合，フレームの送出動作に移行します．しかし，別の端末がほとんど同じタイミングで同じ動作を行っている可能性は否定できません．Ethernet上の端末は，それぞれ独立して動作しているため，ほとんど同じタイミングでデータの送出を行おうとする端末が複数台あることは高い確率で存在します．2台の端末が同時に送出動作を行おうとして，ネットワーク上のデータの存在を確認したときに，ネットワーク上にデータがなければ，2台とも送出動作を開始してしまうとい

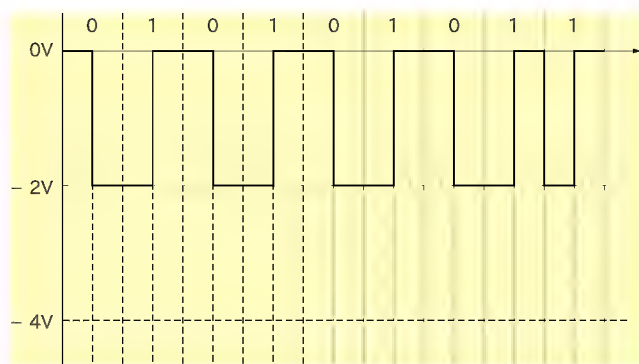


図3　ネットワーク上にデータが存在する場合のケーブル上の信号例（10Base-5）

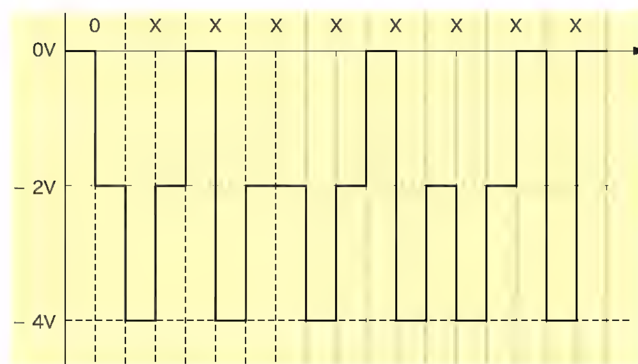


図4　送出データが衝突した場合のケーブル上の信号例（10Base-5）

うわけです．そしてネットワークに接続された端末の台数が増えれば増えるほど同じタイミングで送出動作が行われる確率は増えていきます．ネットワークを介したフレームのやりとりは電位差による二値情報で行われるので，複数の端末が同時にフレームの送出を行った場合，フレームが衝突し，情報は破壊されてしまいます（図4）．

このようなフレームの衝突を検出するようにしなければ，端末はフレームを送ったつもりでも，実際には役に立たないものがネットワーク上に存在することになり，目的を果たせません．受け取るほうからすれば，相手先のアドレスも破壊されているので受信できませんし，受信途中から衝突が発生した場合は，途中までは正常な情報であったとしても，Ethernetフレームの最後にあるFCS（Frame Check Sequence）でエラーとなり，破棄することも可能です．しかし，送るほうからすれば，衝突の検出を行っていなければ送出は完了していると考えてしまうので，こちらは問題が生じてしまいます．TCPのような上位のプロトコルでは情報の再送制御も可能ですが，Ethernetフレームでは，フレームひとつひとつが，基本として独立した情報単位とされるので，受信側からの再送要求はありえません．そのため，送り側は衝突の検出を行い，ネットワーク上において，送出したフレームが確実に存在することを見極め，再送処理を行う必要があります．

上位のアプリケーションから見ればともかく，衝突の発生自体は，Ethernetからすればレイヤ2の動作としては正常な動作の一つなので，受信側から指示が出るのはおかしく，送信側が責任をもって処理しなければなりません．

このことから，Ethernetとして基本的な通信を行うことができる範囲は，衝突を検出できる範囲となってしまい，この範囲をコリジョン・ドメインと呼びます（コラム1）．

衝突を検出した場合，送信を途中で取りやめ，再送処理に移行するということはむだな情報がネットワーク上に存在する時間を短縮し，ネットワーク・リソースのむだを軽減するという観点からも正しい動作となります．

ただし，衝突が検出されたといって送出中のフレームを停止したとしても，即座に再送処理を開始するわけにはいきません．理由は簡単で，フレームの衝突を起こして，送出を取りやめた複数の端末が，その直後，お互いにすぐに再送を行うと，まちがいなく再送フレームも衝突を起こすからです．

4 バック・オフの話

フレームの送出において衝突が発生した場合，仮に即座に再送処理を行わなかったとしても，同じ動作を行っていれば，どれくらい時間を置こうが再度衝突してしまいます．つまり，衝突検出後10ms待機の後に再送処理とするならば，衝突を起こしたフレームを送出した2台の端末から，結局10ms後にフレームが送出され，再び衝突が発生します．

この再送処理について，インターフェース単位で設定を行い，あるインターフェースは1ms後，別のものは3ms後とすると，再度の衝突は発生しませんが，通信を行う機会が均等に与えられないことになるので問題となります．

よって，ここでは乱数が用いられます．衝突の発生した回数によって決められる範囲内の値を乱数によって決定し，あらかじめ決められた時間幅のタイム・スロットに対して，何タイム・スロット経過した後に再送処理に戻るかを個々のインターフェースで決定するようにしています．

この待機時間をバック・オフ・タイムといい，詳細は第2章で解説します．

5 Ethernetフレームの構成と各部の役割

Ethernetフレームの分類は大きく二つあります．形状そのものは1種類といってもよいのですが，一般的には2種類に分類されています．一つはEthernet IIフレームと呼ばれるもので，もう一つがIEEE802.3フレームと呼ばれるものです（図5）．

Ethernetフレームの構成は，フレームの先頭からプリアンブル（Preamble），レイヤ2ヘッダ，データ部，フレーム・チェッ

column 1

衝突の検出機能とそれによる制限

● 衝突検出と最小パケット・サイズ

当然のことなのですが，信号の伝播に利用される電気や光にも速度があります．つまり，端末から送信された情報も，即座に受信側に届くのではなく，距離に応じてそれなりの時間を必要とします．この速度ですが，実際はメタル・ケーブルにおける電気信号でも光ファイバ内の光でもたいした差はなく，真空中における光速（3×10^8 m/s）の0.55～0.6倍程度の速度で伝播します．これは100m進むために，およそ600nsかかることになります．

しかし10Mbpsというインターフェースにおいて1ビットを表す時間幅は100nsなので，たとえば500mの同軸ケーブルの端にある端末Aからフレームが送出される場合を考えると，端末Aが20ビット目を送出していても，実際には反対側にある端末Bには先頭のビット情報さえも届いていません．ということは，端末Bから見ればネットワーク上には空きがあるように見えるので，データの送出を開始しても問題はないと判断します．しかし，数百ns後にはネットワーク上でフレームの衝突が発生します．

そして，この衝突したという情報も同じ時間をかけて，送出側の端末Aまで伝播されます．この衝突したという情報が端末Aに到着するのは，送出中のフレームが送出完了しないうちに届かなければなりません．フレームの送出が完了したあとに衝突情報が到着した場合，送出したフレームによるものなのか，それともまったく関係ない端末による衝突情報なのか判断できなくなってしまうからです．

フレームが長い分には問題はないのですが，Ethernetにおいて用いられるフレームで最小のものは64バイト（512ビット）なので，この大きさのものが送出完了するまでに衝突情報を検出できるネットワークの大きさは，ケーブル間をつなぐためのリピータによる通過遅延も考慮に入れて2500m（同軸ケーブル5本とリピータ4台まで）となります．このように最短のフレームを用いた場合でも，フレームの送出を完了させる前に衝突を検出させることができる範囲をコリジョン・ドメインといいます．つまり，Ethernetのメカニズムのみに頼って通信を行うことができる範囲は2500mが限界ということです．

この範囲を超えて通信を行おうとする場合には，中継を行う機器を配置し，その中継装置によってEthernetによる通信を一度終了させる必要があります．中継装置によってフレームを取り込み，それ以降の通信に関して衝突による再送を代行させるように構築すればよいことになります．

● 通信速度の向上とコリジョン・ドメイン

話は変わって，最近のインターフェース速度の向上には目を見張るものがあります．バス接続形状であった同軸ケーブルが使われなくなり，ツイスト・ペア・ケーブルを用いたスター型の接続となったため，ネットワーク構築の自由度が高くなるとともに速度も向上しました．インターフェースの速度が向上するということは，1ビットの情報を送出するために必要となる時間が短くなるので，同じ大きさのフレームを送出するために必要となる時間は短くなるということを意味します．

つまりFast Ethernet（100Mbps）のインターフェース速度は，Ethernet（10Mbps）の10倍なので，同じ大きさのフレームの送出に必要な時間は10分の1になり，それにともないコリジョン・ドメインの大きさも10分の1程度（200m）になってしまいます．ギガビットEthernetになれば，さらに10倍なので，コリジョン・ドメインはなんと25mになってしまいます．Ethernetのメカニズム（CSMA/CD）だけで通信を行うことができるネットワークの長さは，25mが限界ということです．これはツイスト・ペア・ケーブルの制限長である100mさえクリアできていません．100mのツイスト・ペア・ケーブルで接続されたギガビットEthernetの端末が，最小フレームを送出した場合，先頭のビットが中継のハブに到着したときには，端末はすでにフレームの送出を完了してしまっていることを意味します．

ギガビットEthernetでは，この問題に対応するためにキャリア・エクステンションといって，フレームの送出が実際には終了していても終わっていないように見せかけるという手法を導入し，見かけ上の最小フレームの大きさを512バイトとしました．そして，さらにむだをなくすために複数のフレームを一つに見せかけるフレーム・バーストという手法も導入しました．

それでもようやく8倍の200m程度になったにすぎず，さらに10倍の速度をもつ10ギガビットEthernetではキャリア・エクステンションを用いてさえも20mまで短くなってしまいます．キャリア・エクステンションを用いなければ，ネットワークの長さはわずかに2.5mしかありません．もはやネットワークとは呼べない代物になってしまっています．

このため，ギガビットEthernetにおいては衝突が発生するネットワーク構成の場合にはキャリア・エクステンションを用い，衝突が発生しないような構成であれば従来と同じ64バイト・フレームを利用するようにしています．

さらに10ギガビットEthernetでは，とうとうEthernetの代名詞でもあったCSMA/CDを利用しなくなってしまいました．「10ギガ・インターフェースでは，CSMA/CDを用いていないのに，どうしてEthernetと呼ぶんだ？」，「それはIEEEの802.3で検討されているからだよ」などと冗談が出るほど本末転倒とも思える状況となってしまっています．

ク・シーケンス（FCS：Frame Check Sequence）からなっています（ちなみに，ここでの表現は説明のための用いているので，プリアンブルの次の領域をレイヤ2ヘッダなどと書いているものはあまりないと思われる）．

● プリアンブル

これはフレームの先頭8バイト（64ビット）をさします．内容は単純で31回分'10'というビット列を繰り返し，最後に'11'が続きます．IEEE802.3フレームでは，先頭から7バイト（56

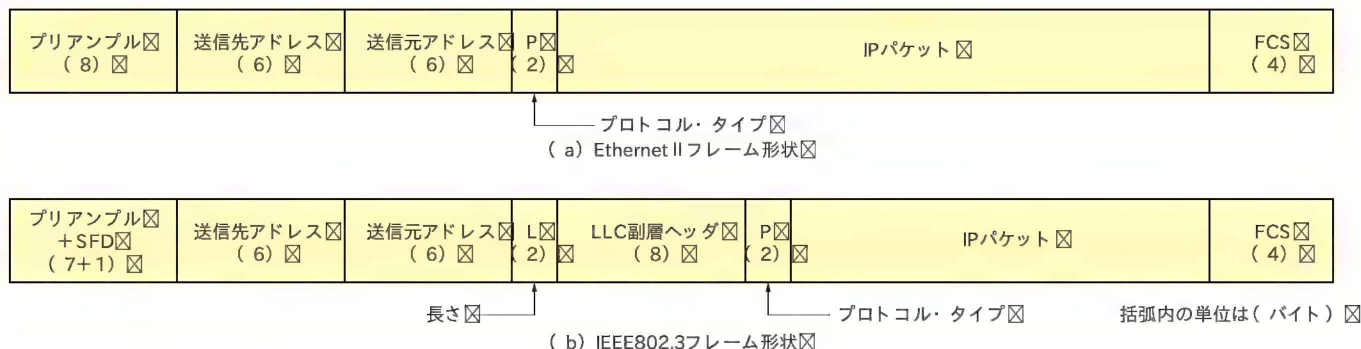


図5　Ethernetのフレームの構成

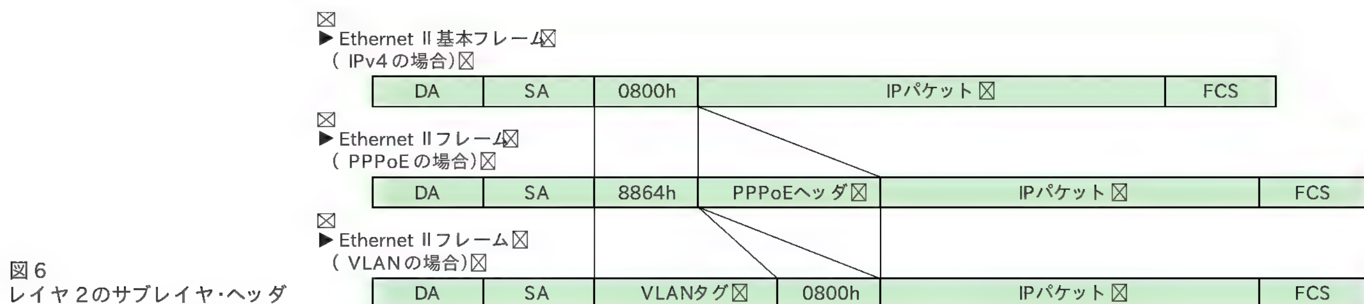


図6
レイヤ2のサブレイヤ・ヘッダ

表1　長さ/プロトコル・タイプ

0000～05DCh	IEEE802.3 Length
0800h	IP(DOD IP：Internet Protocol)
0806h	ARP(Address Resolution Protocol)
8035h	RARP(Reverse ARP)
8037h	IPX
809Bh	EtherTalk(AppleTalk over Ethernet)
86DDh	IPv6

ビット)のみをプリアンブルと呼び，最後の8ビットをSFD(Start Frame Delimiter)と呼ぶようになっていますが，ビット列で見たときの中身はEthernet IIフレームであろうが，IEEE802.3フレームであろうが同じものです．

Ethernetはそれぞれの端末が独自に動作するため，フレームの送出タイミングは送出側のクロック・タイミングに依存し，受信側はそのタイミングに同期しなければなりません．このためプリアンブルを用いて受信のためのタイミングを取ります．

● レイヤ2ヘッダ

次にレイヤ2部分の情報が入ったヘッダが来ます．この部分の大きさは可変長で，状況によって変わります．

レイヤ2ヘッダでも，とりあえずはEthernet IIとIEEE802.3に分類されますが，どちらにせよ先頭から12バイトは同じです．ここには6バイト(48ビット)ずつの，送出先アドレス(DA：Destination Address)と送出元アドレス(SA：Source Address)が入ります．このアドレスはMAC(Media Access Control)アドレスといい，世界で二つと同じ番号がないように管理されています．このアドレスを用いてインターフェース，しいては端末の識別を行うようになっています(コラム2)．

次に来る2バイトがEthernet IIフレームとIEEE802.3フレームの識別を行う部分です．IEEE802.3では，ここはLengthと呼ばれ，これ以降の大きさを示しています．ヘッダ部を除くフレームの最大長が1500バイトなので，1500を示す05DChまでの値が入っていた場合は，IEEE802.3フレームとして認識し，その後にLLC(Logical Link Control)ヘッダが3バイト続きます．そしてそのあとにIPパケットなどのレイヤ3のデータが続くことになります．

Ethernet IIでは，ここはタイプと呼ばれ，この部分の次に来る情報の形式を示します．次にIPパケット(IPv4)が来るような場合は，IPv4を示す0800hが格納され，以降IPパケットのデータが続きます．ちなみにVLANを用いる場合であればVLANタグはタイプの前に入ることになります．なお，その後IEEE802.3でもタイプを用いてもよいことになり，この部分は長さ/プロトコル・タイプとして用いるようになっています(表1)．

また，PPPoEなどを用いるような場合であれば，タイプではPPPoEを示す情報を格納の後，PPPoEヘッダなどが続き，そのあとにレイヤ3のIPパケットなどが続きます(図6，MPLSのタグ情報やPPPoEのヘッダは分類上レイヤ2に含まれる)．

● データ部

ここにはIPパケットなどの上位レイヤの情報が格納されます．

● フレーム・チェック・シーケンス

ここはレイヤ2ヘッダとデータ部に格納された情報が正常で

News Flash――アトミックス，磁気回路部分の効率化に有効な材料であるアモルファス合金粉末を量産
(株)アトミックスは，電源コアなどの磁気回路部分の小型化および効率化に有効な材料であるアモルファス合金粉末の量産を開始した．現在は月産10万トンだが，12月末には月産50万トンを目指している．

column 2

MACアドレス

MACアドレスはIEEEが管理している番号体系で，インターフェースに(半)固定されています．IPアドレスと異なり，運用形態にあわせて変わることはありませんし，そもそも利用者や運用者のつごうで変えられるものでもありません．

MACアドレスは48ビット(6バイト)で構成され，上位3バイトがベンダ・コードとして，そのインターフェース・カード(もしくはポート)を作った会社に引き渡されます．多くの製品をリリースしている企業には，機種単位でベンダ・コードを取得しているところもあります．

下位3バイトは，その会社もしくは機種単位で管理されており，世界中に二つと同じ番号が存在しないようになっています…とりあえずは．しかし，現実には，1台1台違うからと回路を変えるわけにもいかないので，通常は出荷時にフラッシュ・メモリなどに書き込むようになっている製品が多く見られます．いってみれば製造番号のようなものとして扱われるのですが，時折ここでミスが発生します．とくに同一シリーズの製品を同時にラインに流す場合などに指示ミスが生じ，確認があまいと同じMACアドレスをもつ機器が市場に出てしまうことになりかねません．

MACアドレスはレイヤ2で用いられるため，隣接ノード間(具体的にはハブで接続された範囲と思っていれば問題ない)での接続でしか用いられないので，Ethernetフレームがやりとりされている範囲内に同一アドレスの機器が存在しない限り問題動作は起こさないため，万が一ミスにより同じMACアドレスをもった機器が数台市場に出たとしても，まったく気がつかないまま，その機器が人生(機器生?)をまっとうしてしまうこともありえます．

しかし最近では，レイヤ3であるIPアドレスの一部にMACアドレスを使用するというものも出ています．現時点で一般的に利用されているIPv4アドレスは，そもそも32ビットしかないので48ビットあるMACアドレスを格納しようにも，入りません．MACアドレスを利用しようとしているのはIPv6のアドレス体系においてです．

IPv6アドレスは，128ビットあり，上位64ビットがネットワークとして見た場合に上位に位置するルータから端末の位置情報として渡されます．これをプレフィクスといいます．そして下位64ビットにMACアドレスを格納します．ただしこの場合，16ビット分だけMACアドレスのほうが小さいので，MACアドレスをベンダ・コードとベンダが管理するエリアに分割し，その間に`FFFEh`を挿入します．

もともとMACアドレスは世界中で重複のないアドレスであるため，それを利用してIPアドレスを生成すれば，当然その番号は世界中に重複のないものとなります．しかし番号体系に位置的な情報がないため，それをそのまま用いるとドメインもへったくれもないので，ルータに膨大な(というか致命的な)負荷が生じてしまいます．このため，アドレス上位に位置情報を付けることでルーティングの処理が行えるようになっています．

ちなみにIPv6アドレスは，IPv4に比べてはるかに広いアドレス空間をもっていますが，アドレス配布の観点から見れば，ちょっと大盤ぶるまいをしすぎているようにも感じます．つまりMACアドレスが重複しない以上，すべてのプレフィクスにおけるIPv6アドレスをリザーブしていることになります．ということは，一例を示すと，筆者が現時点で利用できるIPv4アドレスはプロバイダから借用している一つだけなのですが(これもアドレスを契約上指定して確実に使えるというものではなく，事実上筆者が占有しているという程度に過ぎない)，IPv6アドレスで見ると，筆者はMACアドレスをもつものを10台持っているので，全プレフィクス×10個のIPv6アドレスがすでに筆者が占有できるものとなっています．

IPv6アドレスにおいて，IPv4アドレスで騒がれているように枯渇が発生するとすれば，それはIPアドレスとしてではなくMACアドレスの枯渇として先に表れます．現時点におけるMACアドレスの配布状況を見てみると騒ぐような状況ではないのですが，IPアドレス以上に回収のできない配布体系なので，そのうち問題になってくるかもしれません．

不要になって廃棄するEthernetインターフェースのカードなどがあった場合は，MACアドレスを控えておくと，将来何かの役に立つかもしれませんね(そのMACアドレスが使えるかは保証しないが)．

あるかどうかの確認を行うために用いる値が格納されます．受信したフレームについて，このFCSを用いて受信した内容が信頼できるかを確認し，エラーであることが判明した場合，フレームの廃棄を行います．エラーではなく，受信したフレームが正常であると判断された場合，FCSとレイヤ2ヘッダ部を取り外して，データ部を上位の処理を行う部分に引き渡します．

6 Ethernetの種類

実はEthernetと呼ばれるインターフェースにはいくつもの種類があります．時代とともにネットワークを利用する環境は変化しているので，技術もその時代の要求にあわせたものを実現するために発展します．

Ethernetの基本となるべき10Base-5を見ると，確かに通信のメカニズムはすばらしいものですが，運用面を考えた場合，タップ・トランシーバの取り付けなどに技術を要したため，だれにでもできるというものではありませんでした．そのため，端末の接続を容易にするために10Base-Tが市場に登場しました．

容易に利用できるようになると，必然といってもよいくらい端末台数が増加します．すると，ネットワーク容量の不足が問題になったので，大容量のネットワークと，それに見合ったインターフェースが必要になり，100Base-TXが現れました．これに関しては10/100Base-TXと書かれているものをよく見かけると思いますが，この表現は10Mbpsインターフェースと

100Mbpsインターフェースを自動(もしくは手動)で，相手側のインターフェースに合わせて動作させることができるというもので，これによって100Mbpsインターフェースである100Base-TXへの移行がスムーズに行われました．ほかに長距離接続が可能なインターフェースの必要性や，より高速なインターフェースの必要性により多くの種類が登場しています．

数多くあるEthernetインターフェースを正確に識別するために「○○Base-xyz」という表記が用られます(図7)．

先頭の○○はインターフェースの速度を表していて，基本としての単位はMbpsです．このため10Mbpsのインターフェースであれば"10Base-△△"という表記になります．現在のところ，唯一の例外は10Gbpsのインターフェースをもつもので，この場合のみ単位はbpsとなり"10GBase-△△"となります．Mbpsを基本として10000Base-だとか10KBase-のようにはなり

column 3

Broad-Bandとブロード・バンド

ちまたで話題のブロード・バンドですが，エンド・ユーザに対して，そこで提供されているインターフェースはおもに10/100Base-TXです．つまりEthernetの分類においてはBroad-Band方式ではなくBase-Band方式のインターフェースを用いています．では，なぜそれがブロード・バンドなのかといえば，ちまたで騒がれているブロード・バンドに対応することばは，Base-Bandではなくナロー・バンド(狭帯域)だからです．ですからブロード・バンドは「広帯域」と訳されます．

EthernetでいうBroad-BandとBase-Bandの違いは，伝送路上でやりとりされる伝送ペアが複数本あるか，それとも1本かで分類されています．Broad-Bandの場合，同時に複数の通信を並行して行うことが可能であり，それぞれに，独立したサービスを割り当てることが可能となっています．どちらかといえばケーブル·テレビのシステムに似ているものです．つまりケーブル内の情報の流れが，1本か複数本かという意味でもあります．ただ，ここでいう1本とはEthernetとしてであって，当然その上位としてVoIPやHTTPのアプリケーションを同時に使用するということは可能です．

さて，狭帯域に対する広帯域としてのブロード・バンドですが，こちらは細いか太いかという判断となるので，どこで分けるかということについては実は明確な基準はありません．1.5Mbps以上あれば十分という人もいれば，45Mbps以上はなければ広帯域とはいえないという人もいます．

筆者がISDN用のターミナル·アダプタを開発していたころは，2MbpsのH12インターフェースは狭帯域に入れられていたように記憶しています．最近はあまり聞かなくなりましたが，一時流行(はや)ったマルチメディア通信が実現可能なインターフェースとして提供されていれば,「広帯域という意味でのブロード・バンド」と呼んで差し支えないのではないかと個人的には思っています．どうせ時代が変わればことばの意味も変わるものですし．要は，現在ブロード・バンドと呼ばれているアクセス回線によって,「今までにはないような新しいサービスが今後も登場してきますよ」といった程度に考えていればよいのではないでしょうか?

ここで少々話をBase-Bandに対するBroad-Bandのほうに戻しますが，Base-Band方式は一つの流れがあり，Broad-Band方式は複数の流れが存在していると書きました．しかし，たとえば1000Base-LR4といったDWDM方式を用いたインターフェースとの違いはどこにあるのでしょうか?

Base-Band方式に対する方のBroad-Band方式は，1本のケーブル上に周波数(もしくは波長)が異なる複数の伝送路を用意するというものです．1000Base-LR4などに用いているWDM(Wavelength Division Multiplexing)はといえば，1本の光ケーブル上に波長の異なる複数の伝送路を用意するというものです．つまりこの部分だけを比較すると同じものになってしまいます．なぜ，名称を1000Broad-LRとしないかといえば，レイヤ1の信号形態はともかく，伝送できる情報の数が異なっているためです．

Broad-Band方式のほうは，ケーブル·テレビと同じ方式と先に書きました．つまり，異なった周波数による伝送路にそれぞれに別の情報が流れているのです．このため，複数の情報を複数の伝送路を用いて同時に伝送することが可能です．1000Base-LR4のほうはといえば，一つの情報を複数の伝送路に並列にして伝送しているのです．つまり一つの波長だけを見てみると，その伝送路の伝送能力は1000Mbpsもないのです．四つに分割された伝送路の合計の伝送能力が1000Mbpsなのです．

最近では，技術的にはEthernet上もしくはIP上で，ほとんどのアプリケーションに関する情報を並列に伝送可能であるため，Broad-Band方式はあまり必要とされませんが，新しいアプリケーションに対応するために，広帯域という意味でのブロード・バンドの重要性はますます増しています．

蛇足ですが，なぜ「常時接続」というかについては，筆者はいまだに理解できていないので説明できません．機能や信頼性から見た常時接続性では，固定電話のほうがはるかに高い(最近はちょっと疑問もあるが)わけなので，後から現れた割に不安定なADSLやCATVによるインターネット·アクセスが使ってよいことばではありませんし,「常時接続していてもだいじょうぶ」という意味であったとしても，現状のNATが入っているとネットワーク側からの着信もままならないような状況では，やはり固定電話に劣ってしまいます．通信を行う際のシーケンスを比較しても騒ぐほどの違いは見当たりません．しいてあげれば，UDPパケットをいきなり送信できるという違いくらいのものですが，そのようなアプリケーションなどそうそう目にするものではないので，何を目的としたことばであるのか，現状でも理解できていません．

ちなみに筆者は固定電話を利用する際に，いちいちケーブルをつなぐようなことはしておらず，常時，接続したままのものを利用しています(笑)．

ません．

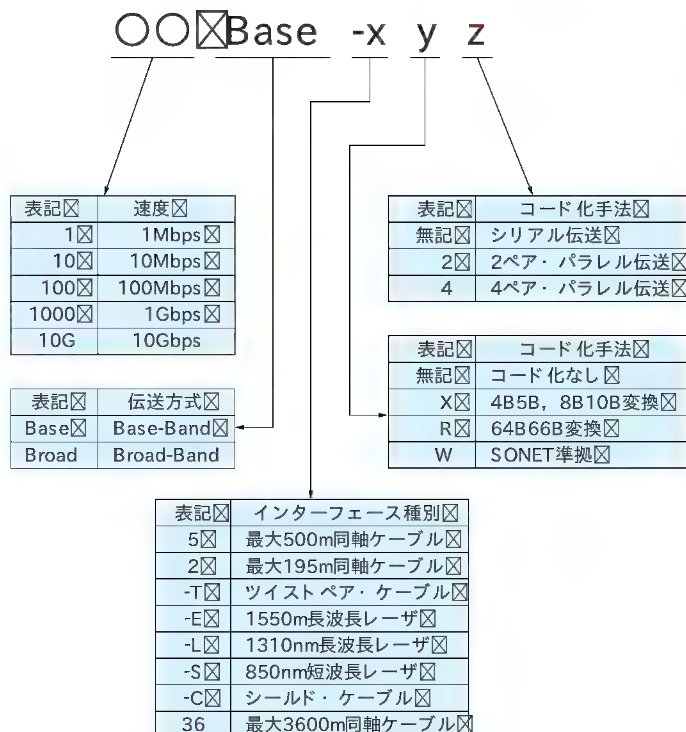
なお，ここの速度はEthernetフレームを伝送するための速度で，インターフェース上でやりとりされるビット列の速度ではありません．たとえば100Base-TXでは，Xがついているので4B5B（4ビットを5ビット・コードに）変換を行ったうえで100Mbpsの伝送を可能なようにするため，ビット単位でみた伝送路の能力としては125Mbpsとなっています．

次のBaseは伝送路の信号伝播の方式がBase-Band方式を用いていることを意味しています．Base-Band方式以外にはBroad-Band方式（インターネット常時接続などと違訳されているものではない点に注意）があるのですが，実際の種別としてはほとんどなく，10Broad36というものが昔あったなぁという程度のものです（よって10Base-Tなどを10BTと略記するのは，Broadでも頭文字はBなので本当は正しくない（コラム3））．

最後のパラメータは，インターフェースにおけるレイヤ1の特徴を示しており，現状最大3桁で表示されています．なお，選択肢や特徴のない項目については省略できるようになっています．

よって，たとえば10Base-Tであれば，「10Mbpsのフレーム伝送速度を持つBase-Band方式のインターフェースで，ケーブルにツイスト・ペア（T）を用いるものである」ということがわかりますし，10GBase-ERであれば「10Gbpsのフレーム伝送速度を持つBase-Band方式のインターフェースで，1550nmのレーザ（E）を用いて，かつ64B66B変換のコード化（R）を行ったものである」ということがわかります．

ちなみに，これらのパラメータの前にある“-”についてですが，実際には10Base-Tから登場しており，それ以前の10Base-5などは，「10Base5」と書いているほうが実は正解です．したがって，正確にはBase（もしくはBroad）のあとに“-”がない場合は，そのあとにケーブル長を示す5（500m），2（195m），36（3600m）などが来ることになり，“-”があったあとに来る数字はケーブル長ではありません．たとえば100Base-T4というインターフェースの最後の4は，400mというものではなくツイスト・ペア・ケーブルの四つのペアすべてを用いているという意味になっています．



表記	速度
1	1Mbps
10	10Mbps
100	100Mbps
1000	1Gbps
10G	10Gbps

表記	伝送方式
Base	Base-Band
Broad	Broad-Band

表記	インターフェース種別
5	最大500m同軸ケーブル
2	最大195m同軸ケーブル
-T	ツイストペア・ケーブル
-E	1550nm長波長レーザ
-L	1310nm長波長レーザ
-S	850nm短波長レーザ
-C	シールド・ケーブル
36	最大3600m同軸ケーブル

表記	コード化手法
無記	コード化なし
X	4B5B，8B10B変換
R	64B66B変換
W	SONET準拠

表記	コード化手法
無記	シリアル伝送
2	2ペア・パラレル伝送
4	4ペア・パラレル伝送

図7　Ethernetの種類

なお，どういうわけか，ギガビットEthernetの1000Base-Tに関してのみ，4ペアすべてを使用しているにもかかわらず1000Base-T4とは書きません．このため1000Base-Tだからと2ペアしか結線のないUTPケーブルを使用した場合，通信はできないので注意が必要です．

まつもと・のぶゆき　（株）タムラ製作所

第2章



ハブでポートを増やし，ツイスト・ペア・ケーブルで接続可能な

10Base-Tの詳細

松本 信幸

Ethernetの概要を理解したところで，実際に設計・製作するインターフェースである10Base-Tについて，その詳細を解説する．Ethernetフレームの詳細，FCS，マンチェスタ符号，衝突検出と再送処理など，10Base-Tで使われる技術をよく理解してほしい．

（編集部）

1 10Base-Tというもの

第3章で設計･製作してみようというインターフェースは，現在でもよく使用されているものの中で，もっとも基本的な技術である10Base-Tです．

10Base-Tは，その名称どおり10Mbpsの伝送能力をもったベースバンド方式のインターフェースで，ケーブルにツイスト・ペア・ケーブルを用いるものです．名称の最後の部分にコード化を示すアルファベットがついていないので，コード変換は行わないことを意味し，ビット列による速度も同様に10Mbpsとなります．

名称から読み取ることのできる特徴はここまでなのですが，実際にディジタル情報の通信を行おうとした場合には，ビット情報を正確に伝送するために二つの重要な項目と，それにともなう補足事項が決まっていなければなりません．それが符号化と電気特性です．そしてさらに，ビットの集合によるビット列から意味を見い出し，そこからデータを取り出すために送受信双方で決められた手順が必要となります．この手順であるルールを，その守備範囲に合わせて分類しているのがOSI 7レイヤです．Ethernetでは，この中から特にレイヤ1とレイヤ2について規定しています．

表1　10Base-Tの規格（10Base-Tと100Base-TXとの比較）

	10Base-T	100Base-TX
コネクタ形状	RJ-45	RJ-45
送信ペア（端末側）	1～2番ピン	1～2番ピン
受信ペア（端末側）	3～6番ピン	3～6番ピン
信号形式	マンチェスタ	MLT-3
信号速度（Mbps）	10	125
データ速度（Mbps）	10	100
クロック速度（MHz）	20	25
クロック精度（%）	±0.01	±0.01
コード化	―	4B5B
使用フレーム		
Ethernet II	○	○
IEEE802.3	○	○
最大フレーム長（※）（バイト）	64	64
最小フレーム長（※）（バイト）	1518	1518
フレーム間ギャップ（μs）	9.6	9.6

※：プリアンブルは除く，FCSは含む

2 10Base-Tの規格

10Base-Tのレイヤ1で示されるおもな項目としては，インターフェースのコネクタ形状やピン・アサイン，伝送速度，符号化形式，クロック処理，衝突処理（衝突検出，処理），キャリア検出（ネットワークの空き状態確認）などが挙げられます．表1に，参考のため100Base-TXと比較した10Base-Tの概要を示します．

そして，レイヤ2のおもな項目としては，フレームの生成と分解，受信フレームの正常性確認，競合処理（衝突発生時の再衝突回避）などが挙げられます．

3 10Base-Tの送信動作

さて，ここからは10Base-Tにおける動作について，データの流れに沿って紹介します．

● フレーム生成

10Base-Tというか，Ethernetの守備範囲であるレイヤは1と2なので，10Base-Tの機能ブロックに到着してくる情報は，たとえばIPパケットのL3_PDUなどと呼ばれるものです（図1）．ちなみに今回は，IPパケットなどのレイヤ3情報をパケット，Ethernetフレームなどのレイヤ2情報をフレームと呼んでいます．

この情報を，Ethernetフレームとして送出するためには，パケット（L3_PDU）の先頭にプリアンブル，送信先MACアドレ

ス(DA)，送信元MACアドレス(SA)，Ethernetタイプをつけ，さらに最終部分にFCSを付与することになります．

プリアンブルは固定長/固定値なので，フレームの送出開始時に事前に用意したものを送出すればよいだけなので，大したことはありません．

送信先MACアドレスは，隣接している端末，すなわちMACアドレスが見える範囲に存在する端末を指定する場合には，その端末のMACアドレスが入りますが，ルータなどを越えていかなければならない直接見えない端末に対する場合には，おもにデフォルト・ゲートウェイとして指定されている機器のMACアドレスが格納されることになります．ただし，経路制御など，ネットワーク状態確認などに用いる場合においては，ブロードキャスト・アドレスが格納されることもあります．

送信元MACアドレスは，自らのインターフェースに与えられたアドレスが格納されます．

そしてEthernetタイプは，続くパケット(L3_PDU)が何であるのかを識別する情報が格納されます．もっともポピュラなIPv4であれば，`0800h`となります．Ethernetタイプの後にパケットが続いて，その後にFCSが続きます．

● FCSの生成

FCSには，プリアンブルを除いたEthernetフレームが，受信側において正常に受信されたかどうかを確認するために用いるチェック用の値が格納されます．

FCSの値の計算方法は，生成多項式を用いた巡回符号で求められます．巡回符号に用いる生成多項式には，いくつかの種類があります．この使い分けは計算範囲であるフレーム(もしくはパケット，セルヘッダ)の大きさによります．Ethernetフレームの場合，最小64バイト，最大1518バイト(ともにFCSを含む)にもなるので，それにともなってFCSも大きくなり，Ethernetでは32ビットとなります．

生成多項式も，CRC32といわれる図2の式が用いられます．この生成多項式を用いて求められた結果が，Ethernetフレームの最後尾に格納され，フレーム生成が完了します．そしてその後の処理を行うためにレイヤ1に引き渡されます．

● ネットワークの空き状態確認

レイヤ1が分担する動作部分にフレームが引き渡されたとき，つまり，レイヤ1担当部分から見て，送信すべきフレームが手元に存在した場合，最初にネットワークの状態確認を行います．

具体的な動作としては，別の端末から送出されているEthernetフレームをインターフェースが受信しているかどうかの確認を行います．10Base-Tで利用するCSMA/CD方式では，ネットワーク・リソースを共有するので，受信しているフレームが存在するということは，ほかの端末がネットワーク・リソースを使用していることを意味しています．つまり，ネットワーク・リソースは使用中であるため，送信動作を行うことができないということになります．

受信中のフレームが存在しない場合，送信動作を開始しますが，もし受信中のフレームが存在していた場合，送信待機状態に入り，受信中のフレームが終了するまで待ちます．なお，受信中のフレームの受信動作が完了しても，即座に送信動作に移行できるわけではありません．

Ethernetでは，フレームの送出タイミングは，送信動作を行おうとする端末のクロックに依存するので，ある端末が送出するフレームのクロック・タイミングと，別の端末が送出するフレームのクロック・タイミングは，規定値(±0.01％)から逸脱しない範囲で異なっているのが普通で，そもそも信号変化を示す位相の関係はまったく独立したものとなっています(図3)．

ある端末が送出したEthernetフレームに対して，間髪入れずに別の端末からEthernetフレームが送出された場合，受信側においては，クロック・タイミングや位相の変化に追従できなくなってしまいます．このため，Ethernet上では，フレームとフレームの間を，一定時間空けるように規定しています．この間隔をフレーム間ギャップ(Inter Frame Gap)といい，10Base-Tでは9.6 μsと決まっています．

こうして，空いていれば即座に，ほかのフレームを受信中であれば受信完了の後，さらに9.6 μs待機した後に送信動作を開始します．

● ビット送出順序

Ethernetのデータ送信時のビット順序は少々変わっていて，先頭にあるプリアンブルは，バイト単位で見た場合，上位のビット(MSB)から送出を行います．しかし，これが宛て先アドレス以降になると，バイト単位で見たときの下位ビット(LSB)からの送出に切り替わります．そして最後のFCSはまた逆になり，上位ビットからの送出となります(図4)．

● 符号変換

ディジタル情報の伝送は‘0’と‘1’による伝送ですが，「何をもって‘0’とするか，また‘1’とするか」ということを定義しておかなければなりません．多くの場合，2本の電線を用いて，

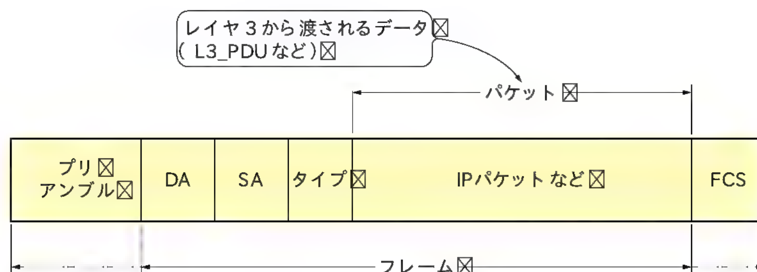


図1 フレームとパケットの定義

$$CRC32 = X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^{8}+X^{7}+X^{5}+X^{4}+X^{2}+X+1$$

図2 Ethernetで使用するCRC生成多項式(CRC32)

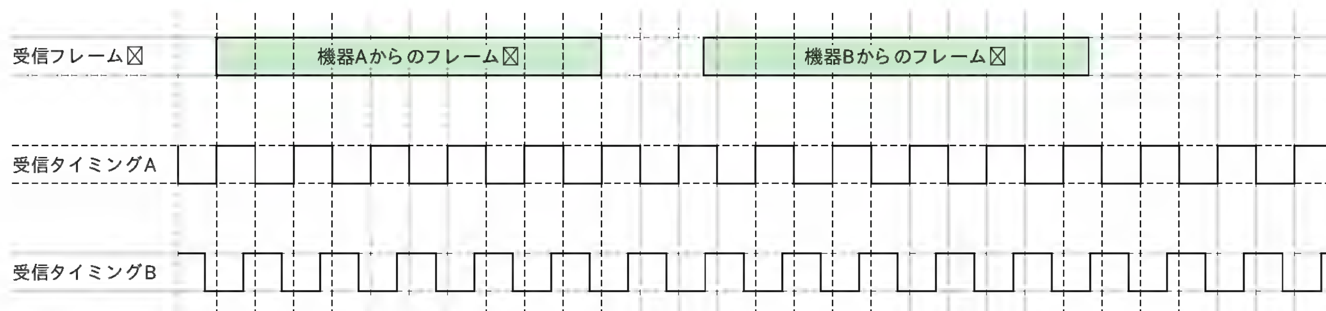


図3　各端末は独立したクロックで動作している

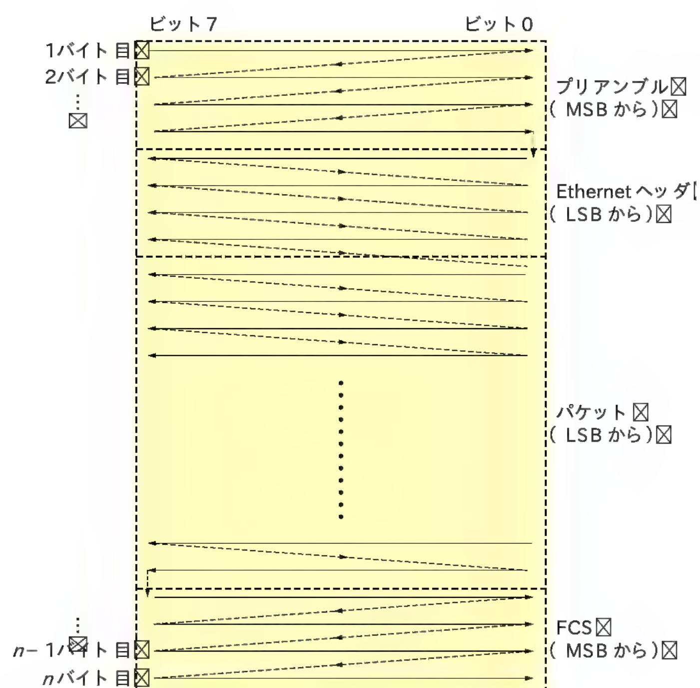


図4　バイト単位データの送出順序

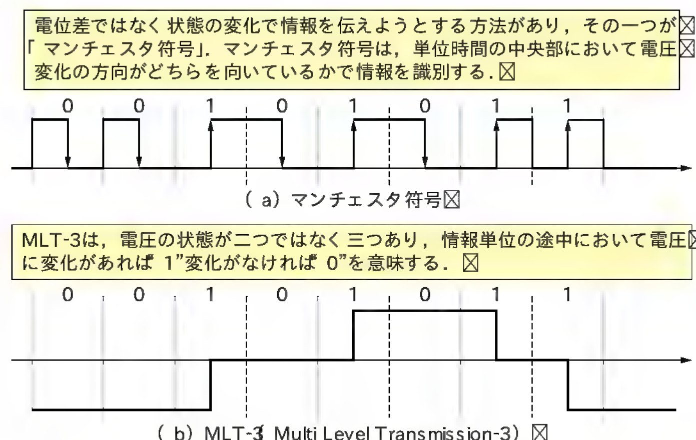


図5　符号方式の違い

基準となる電線(グラウンドとして扱われる)から見てもう1本の電線との間の電圧(電位)が高い状態を'1'，電位差がない，もしくは低い(0Vに近い)状態を'0'として扱われていることが多くあります．しかし，この場合でも「何ボルトをもって『高い』と解釈するか」を決めておかなければなりません(たとえばLV-TTLでは'1'と判定する電位差でも，TTLでは不定領域になっている)．

'0'から'1'，もしくは'1'から'0'への状態の変化にも多少なりとも時間を要するので，'1'として認識するまでの時間が短いほど，高速な信号に対応可能となりますが，その反面，ノイズに弱くなってしまうので，ケーブル長などの使用環境に影響を及ぼします．

さて，'0'と'1'の判断ですが，実際は電圧以外のものを用いることもよくあります．一つには状態の変化を用いるもので，基準となる時間枠内に変化がなければ'0'で，電圧に変化があれば'1'を意味するというものや，基準となるタイミングにおいて，かならず変化は起こるけれども，その変化の向きが，0VからXVへの変化であれば'0'であり，逆にXVから0Vへの変化であれば'1'を意味するというものもあります．

実はこの，'0'と'1'の意味付けは，同じRJ-45コネクタを使い，オート・ネゴシエーションによって自動で速度を切り替えることも可能な，10Base-Tと100Base-TXの間でさえ違うものが用いられています．10Base-Tではマンチェスタ符号が用いられ，100Base-TXではMLT-3(Multi Level Transmission-3)が用いられるのです(図5)．

▶ マンチェスタ符号とは？

10Base-Tで使用されるマンチェスタ符号はマンチェスタ大学で考案されたもので，伝送しようとする情報ビットの中央で，かならず電位が変化する方式です．情報を示すためのタイミングについては，電位差が高いほうから低いほうに変化した場合には'0'を意味し，逆に低いほうから高いほうに変化した場合には'1'を意味します．ここで注意が必要なのは，状態の検出を行うタイミング以外(つまりビット情報の変化点)での変化は無視するという点です．

また，マンチェスタ符号の大きな特徴の一つに，「状態が三つある」ということがあります．よく見かけるNRZ(Non-Return to Zero)形式の，0Vのときに'0'，XVのときに'1'を示す方法

では，‘0’の情報が伝送されているときと情報そのものがないときの区別がつきませんが，マンチェスタ符号の場合は，情報が‘0’であっても‘1’であっても電位差に変化が生じ，その変化によって識別するので，変化がないということは，すなわち情報がないということになります．このため，Ethernetのようにネットワークを共有するような場合，つまりCSMA/CD方式を利用するような場合においては，NRZなどのように連続した‘0’なのか，情報が存在しないのかを区別できないような情報の形式では不都合で，ネットワーク上にデータが存在しない状態を明確に認識できる必要があります．

マンチェスタ符号を用いた場合のもう一つの特徴は，Ethernetフレームの先頭にあるプリアンブルのように，‘0’と‘1’を交互に送出すると，見かけ上半分の(つまり5MHzの)クロック情報として現れることになります．受信側ではこのタイミングを利用して受信のためのタイミングをそろえます．このように‘0’と‘1’を交互に連続して送出する状態をオルタネート・データと呼びます(コラム1)．

ちなみに，10Base-Tで利用されるマンチェスタ符号での電位に明確な規定はありませんが，受信側で$1V_{p\text{-}p}$以上あれば問題ないようです．

▶MLT-3とは

10Base-Tで利用しているマンチェスタ符号と，100Base-TXで利用するMLT-3とではかなり大きな違いがあります．マンチェスタ符号の場合，電位としての状態は二つで，情報の中央においてどちら向きの状態変化なのかで‘0’と‘1’の二値情報を伝送しますが，MLT-3における電位の状態は三つあります．そして状態変化にも違いがあり，‘0’のときは状態が変化せず，‘1’の情報を伝送しようとする場合にのみ電位の状態が変化します．つまり，オール‘0’の情報を伝送しようとすると信号線上の電位は変化しないままとなり，見かけ上情報を伝送していないように見えてしまいます．クロック同期されていない装置間のデータ伝送で，このような状態は好ましくないのですが，100Base-TXの場合，最後にXがついていることから，情報としてのビット列に4B5B変換を行っています．つまり，伝送しようとする情報が連続した0であったとしても，信号変化としては4B5B変換によって連続した0の状態ではなくなるので，問題はなくなります．

● 10Base-Tのコネクタ

10Base-Tで使用されるコネクタは，電話機で使用されているモジュラ(RJ-11/14)を少々大きくしたようなRJ-45を用います．RJ-45は8ピンなのですが，10Base-Tではこのうち1，2，3，6番ピンの4本を使用しています．なぜ，このように中途半端な用いかたを行うのかといえば，RJシリーズがもともと電話網で利用されているということが影響しています．

よく見かける同様なRJシリーズのコネクタとしては，電話機のハンドセット(受話器)の接続に用いる4接点あるタイプと，電話機そのものを接続するRJ-11です．実はRJ-11は，コネクタ的には6接点まで可能であるのに，2接点しか使用していないです．つまり，コネクタ自体は6芯のケーブルまで使えるようになっています．もっとも市販のものでは，接点自体は二つしかないので，ケーブルをつないでも導通はありません．企業内で利用するキー・テレホンでは4接続まで行っているものもあります．

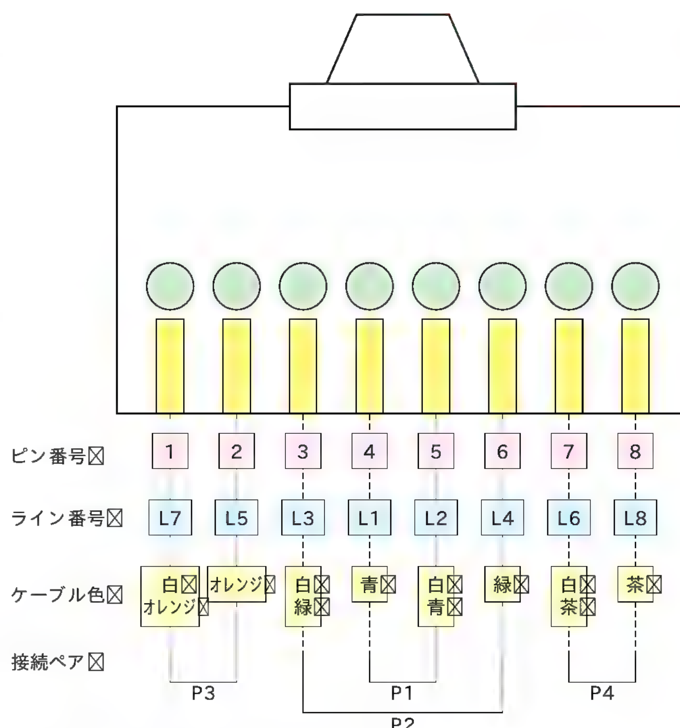


図6　10Base-Tのコネクタのピン配置

さて問題はRJ-11の接続です．本来6芯まで接続できるコネクタで2本しか利用しないのですが，この2本が実は中央から2本しか利用せず，両側に2本ずつ空きがくるようになっています．実際端子の接続を識別するためのライン番号は，中央から外に向かってL1，L2…L6となります．電話機における接続は共通のグラウンドに信号線がくるのではなく，2本一組のペアとなっています．したがって，L1-L2で一つ目のペア，L3-L4で二つ目のペアとなります．

しかし，ややこしいことにピン番号は，端から1，2，3と呼びます．つまり，ライン・ペアとピン番号がまったく異なったルールで並んでいるのです．8ピン・コネクタのRJ-45ではさらにややこしいことに，三つ目と四つ目のペアが両側に存在します(図6)．

10Base-Tではこれらのペアのうち二つ目と三つ目を使用します．つまり，ピン番号でいうところの3-6番ピンのペアと1-2番ピンのペアです．これは電話機で使用しているL1-L2用のペアを避けたうえで続けてアサインするとこうなったというわけです．RS-232Cなどで使用しているD-Subコネクタでは25ピンのコネクタに15ピンのケーブルは接続できませんが，RJシリー

ズでは 8ピンのソケットに 6ピンのプラグは接続できてしまうため，誤接続による障害を防止するために，このようなピン・アサインになっています．もっとも，一部の RJ-48のように外観は非常に似ていて，RJ-45と同じく 8ピンですが，ちょっとした形状の違いにより接続できなくなっているものもあります．

余談ですが，多くの Ethernet インターフェースでは，誤接続が発生して本来とは異なる電圧がかかってもだいじょうぶなように設計されているので，本質的には大きな問題ではありません．しかし，この対応は推奨されているだけでしかなく，必須項目ではないので，やたらと接続することはしないほうが無難です．けっこうよく見かけるんですよ，10/100Base-TX 用の RJ-45コネクタに，ADSL の RJ-11をつなごうとして「通信できない」と言っている人を…（笑）．

● 衝突の検出と上位への通知

ネットワークが空き状態で，フレームの送出が可能な状態にあることが確認された場合，インターフェースはフレームの送出を開始しますが，同様の動作を行っているほかの装置がネットワーク上に存在する場合もありえます．こうした場合，ネットワーク上でフレームの衝突が発生します．つまり送信動作中のフレームは，情報の内容が破壊され，無意味なビット列となってしまいます．

この現象の発生については，インターフェースが，送信動作中においても受信部を確認することにより検出可能です．

ネットワーク上における衝突の発生が確認された場合，インターフェースは送出動作を中断します．また，この場合，受信側で確実に廃棄されるようにするため，中断前にジャム信号を送出し，フレームを無意味な塊にしてしまいます．

フレームの送出を中断したレイヤ 1の機能ブロックは，いったんフレームを衝突発生の通知とともにレイヤ 2に戻します．

● バックオフと送信完了

送信動作を中断したフレームをレイヤ 2に戻す理由は，中断した直後に再送動作としてネットワーク上の空き状態の確認を行った場合，衝突したフレームを送出した別のインターフェースにおいても同様の処理が行われるので，再送動作において再度衝突が発生してしまうことになります．これを回避するための処理を担当しているのがレイヤ 2であるためです．

差し戻されたフレームに対する具体的な処理としては，乱数によって決められた時間分待機状態を継続し，待機時間が終了すると再送開始要求としてレイヤ 1に渡されます．

再送開始要求までの，乱数によって決められる待機時間です

column 1

クロック

情報を正確に伝えるには，送り手と受け手が同じルールに従った動作を行っていなければなりません．そのルールの一つに情報伝送のタイミングがあります．

この情報伝送のタイミングに関する動作の元をクロックといい，通信においては多くの場合，Hz で表します．ネットワークにおいて通信に用いるクロックの分類は大きく自走と従属に分けることができます．

自走とは，ある装置が，自分自身の回路の中に存在する発振器もしくは発振子（を用いた発振回路）によって動作するもので，自走動作している装置間における通信では，送信動作はそれぞれのもっているクロック・タイミングで行われます．Ethernet では，一部の例外を除いてほとんどの場合，この自走による装置間の通信となっています．

逆に従属とは，ネットワークなどにおける装置間において，動作タイミングの親というべきクロック・マスタが存在し，装置間の通信はクロック・マスタから供給されるタイミング情報に従って行われます．クロックの主従関係は，通信を行う 2台の間でマスタかスレーブかを決め，マスタ側のクロックに従ってスレーブ側も送信を行う方式（ギガビット Ethernet など）と，A，B の 2台の通信において，それぞれの機器が別の C に従っている方式（公衆電話網など）の 2通りがあります．

自走と従属の長所，短所としては，自走方式はシステムの構成をシンプルにすることが可能であることや，一部を切り離して運用することが容易であるということが挙げられ，そしてこれはそのまま従属方式の短所の裏返しとなります．逆に従属方式の長所は，システム全体が同一のクロック・タイミングで動作するので，システムがどれほど巨大になっても，装置間の速度差によるデータの破壊や取りこぼしが発生しないことになり，これもまた自走方式の短所の裏返しとなります．

自走と従属の使い分けに関しては，システム（もしくはインターフェース）の速度そのものというよりは，インターフェース間（伝送路上）のデータの密度によります．高速のインターフェースでも実際に伝送しようとする情報がスカスカであれば，自走でも問題ありません．つまり欠点は露見しません．しかし，低速のインターフェースでも情報の密度が濃ければ，取りこぼしや破壊が発生してしまうので，従属動作のほうが良いということになります．

つまり，もともと LAN として使用されていた Ethernet インターフェースを用いて構築するネットワークでは，クロックを要因とするデータの破壊や取りこぼしが発生する可能性があるということです．このような現象は，とくにリピータやスタック・ハブのようなシンプルな機器ほど発生しやすくなります．

現象の説明の前に動作クロックの話をすると，ネットワーク機器はそれぞれ動作クロックの元として水晶発振子や水晶発振器をもっています．これら水晶によるクロックは，たとえば 19.44MHz などのように発振周波数が調整されているのですが，一切の差もなくまったく同じ速度で動作するということはありえません（一瞬同じになるくらいならありうるが）．同じメーカの製品を並べても，部品のばらつきなどによって，わずかでも違いは生じます．それどころか，同じ機器であっても，日の当たるところと日陰，夏と冬でさえ，違いが生じることになります．しかし，違いが出るといって，無視してよいものでもないので，ばらつきやずれの範囲は指定しています．これを ppm

が，51.2 μsを単位時間として，再送回数によって導き出される値を上限とした乱数回時間として規定されています．乱数の上限は［再送開始までの時間単位 $N = 2^X - 1$］で，X が送信にトライした回数です．ちなみに下限は「0」なので，乱数の値によっては待機時間なしで即座に再送動作に移行する場合もあります，［図7(a)］．

衝突を検出して，再送待機になった場合，その動作が1回目である場合，送信動作そのものは2回目であるため $X = 2$ となるので，待機時間の上限は3($N = 2^2 - 1$)となります．このため，乱数により，「即座＝0」，「51.2 μs後＝1」，「102.4 μs後＝2」，「153.6 μs後＝3」のいずれか経過した後，再送処理のため，再度レイヤ1へ渡されることになります．ここで再度衝突が発生した場合，またしてもレイヤ2に戻され，再待機状態に移行します．今度は再送待機も2回目(送信動作としては3回目)ですから，さらなる再送信の開始までの待機時間の上限は，最大7単位時間となり，その範囲(最短：0/最長：358.4 μs)においてランダムに決められます．

ここで注意しなければならないのは，衝突が頻発したとしても，再送待機の時間が単に増えるというものではないという点です．1回目再送では，乱数の幅は3であるため，仮に2単位時間(102.4 μs)待機した後，再送動作を行ったとします．そして，またしても衝突が発生した場合，待機時間の上限は7ですが，あくまで上限なので，乱数の値によっては1でも良く，乱数の目の出かたによっては1回目の再送より2回目の再送のほうが，待機時間が短い場合もありうるということです［図7(b)］．

さらに，何度も衝突を繰り返すからと，無制限に上限値を上げていくわけにもいかないので，X の値には二つの制限があります．一つは待機時間単位の上限値 N を導出する際の制限ですが，この値は10までとなっています．再送回数が10回目になると，再送待機時間の上限は1023まで増えることになりますが，ここでも衝突が発生し，11回目の再送を行うことになったとしても，上限値は1023以上にはなりません．12回目の再送でも同様に上限値は1023であり，この範囲の乱数によって決定されます．

もう一つの制限は，再送そのものを繰り返す回数の制限で，こちらの上限は16回です．再送処理を16回試してもフレームの送出を完了できなかった場合，インターフェースはフレームの送出そのものをあきらめます．

衝突が検出されず，フレーム最後のFCSまで送出が完了し

(Percent Par Million)として表し，10MHz±10ppmというように表記します．この数字は，部品のばらつきはおろか，使用する場所がどこであろうと，夏であろうが，冬であろうが，製品の動作保証する環境下における使用においては，この範囲を外れてはいけないということです．無論，動作保障の上限温度が40℃なのにサウナの中で動かない…などというものは当然相手にしません．

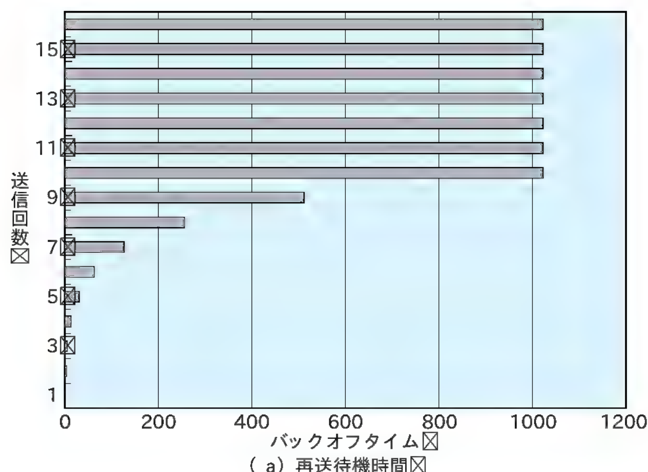
10MHzの10ppmは1Hzです．このため10MHz±10ppmということは9,999,999～10,000,001Hzで動作することを通常の使用環境下(主に0～40℃)において，保証しているという意味です．

さて，話を戻して，データの破壊についてですが，シンプルな機器ほど発生しやすいという理由について，逆のアプローチで説明すると，ルータのような高機能な機器ではフレームの送受信を行うとき，到着するフレームを一度完全に取り込んで，ヘッダなどの情報を解析して処理を行います．Ethernetでは受信タイミングは送信側のタイミングに従いますが，中継装置であっても一度フレームを取り込んでしまうような動作では，あまり問題は発生しません．

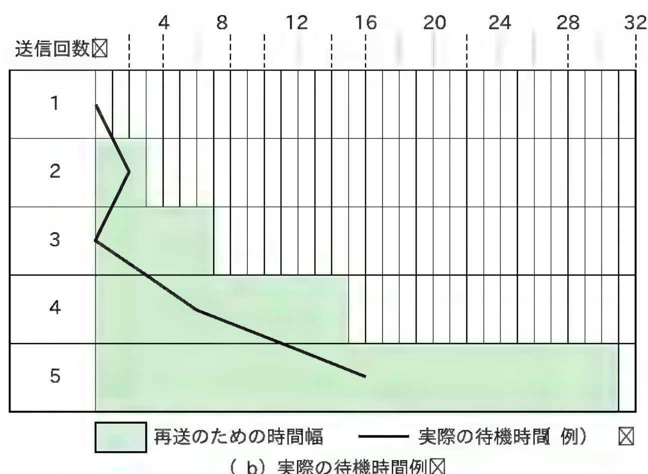
しかし，スタック・ハブやリピータのような機器では，フレームを取り込みながら，ほぼ同時に送出も行います．このときリピータでみると，到着するフレームは送信側のタイミングに従っていますが，中継して送出するためのタイミングは，自分自身の装置内にあるクロックに従います．リピータは，受信したフレームについて数クロック分バッファした状態で送出動作を開始します．リピータの送信クロック速度が送信側装置より遅い場合，受信タイミングに比べて送信タイミングのほうが遅くなります．つまり時間がたつほど，リピータ内にバッファリングされるビットが増えることを意味します．リピータのような安価な装置に膨大なバッファをもたせるようなことはしないため，そのままではやがて溢れることになり，ビットの取りこぼしが発生してしまいます．一度取りこぼしたビットは再現することができないので，このフレームは，最終的な受信側でFCSエラー(もしくはLengthの異常)として廃棄されてしまいます．逆の例では，リピータ内のバッファが空になり，送信しようにも次のビットが到着していないという状態となります．こうなるとフレームが引きちぎられ，同じく廃棄されることになります．これらは，「小さいフレームは問題ないが，ある特定以上の大きさのフレームが通らない」という症状として現れます．

この問題を回避するためには，フレームを小さくするか，バッファ容量を大きくするとともに中継段のバッファを多めに取るといった対策が考えられますが，フレームは最大長1518バイト(Ethernetの場合)と決まっているうえ，バッファを大きく取るということは通過遅延が大きくなるということなので，昨今話題のストリーム系アプリケーションに悪影響を与えることになってしまいます．いちばん簡単な対応は，指定された精度内で動作させるということになります．

ちなみにEthernetインターフェースの精度ではppmは用いません．というのも，Ethernetで規定されているクロック精度はかなり大ざっぱで，±0.01％でよいことになっているためです．つまり，10MHz±1kHzで動作すればよいのです(無理やりppmで書けば10MHz±100ppmとなる)．市販の水晶発振器は安いものでも数十ppmの精度をもっているので，近くのパーツ屋で，名の通ったメーカの部品を買ってくれば，Ethernetインターフェースの回路に使用してもまったく問題はありません．



（a）再送待機時間⊠



（b）実際の待機時間例⊠

図7　衝突検出から再送まで

た場合，フレームの送出動作が完了したとみなされ，次のフレームの動作を開始します．

なお，ネットワークの大きさから考えて，Ethernet の CSMA/CD 動作を正常に行っているならば，たとえば 1500 バイトのような大きなフレームを用いた場合，最終段近くで衝突が発生することはありえません．最小サイズ（64 バイト）分の送出が完了した段階で，それ以降の送出は順調に進むことになります．

もし，万が一，最小サイズを超えた段階で衝突が検出された場合，それは CSMA/CD 方式のルールにしたがっていない端末がネットワーク上に存在するか，ネットワーク構成そのものがルール違反で行われているかのどちらかであると考えられます．

ちなみに，バックオフの基準となる 51.2 μs についてですが，この値は実は最小フレーム（64 バイト）の送出に必要な時間を表しています．10Mbps（10MHz）における 1 ビットの送出に要する時間は 100ns です．そして 64 バイトとは 512 ビットなので，512 × 100ns = 51.2 μs となります．

4 10Base-T の受信動作

● 到着フレームの検出

Ethernet は，ネットワークを共有（シェア）するため，当然，どの端末も使用していないときが存在します．つまり，信号がなにもない状態があります．そしていずれかのインターフェースからフレームが送出された場合に，ほかの装置のインターフェースにフレームが到着することになります．

フレームの到着はオルタネート・データの受信で始まります．つまり，フレームのプリアンブルの部分です．プリアンブルによるオルタネート・データがありがたいのは，マンチェスタ符号におけるビット情報の境界において電位の変化は発生せず，オルタネート・データで示される変化点のタイミングで以降の情報を見ていけばよいからです．タイミングはプリアンブル 62 ビット以内（うち 6 ビットは SFD：スタート・フレーム・デリミタと呼ぶこともある）に同期させるようにしますが，62 ビットの情報については，バッファリングはもとより計数する必要さえありません．読み捨てるだけで十分です．

プリアンブル（もしくはスタート・フレーム・デリミタ）の最終 2 ビットが検出された場合，それ以降の情報を計数しながらバッファリングしていきます．到着した情報であるフレームが終了したとき，すなわち信号変化がなくなったときに，フレームの受信が完了したものとみなします．

そして，ここでフレーム長の確認を行います．受信したフレームの長さが 64 バイトに満たなかった場合は，送信を行っていたインターフェースが，衝突を検出したなどの理由により，フレームの送出を一時中断したとみなし，不完全なフレームとして廃棄します．同様に 1518 バイト以上あった場合においても，何らかの理由でフレームが重なったなどと解釈し，廃棄対象となります．

ここで廃棄対象とならなかったフレームは，レイヤ 2 の処理に引き渡されます．

● フレームの正常性確認

次に受信したフレームが正常かどうかの確認を行います．

現在ではスイッチング・ハブなどの普及で，ノイズや伝送信号の劣化による FCS エラーの発生はあまりありません．しかし，それ以外でも，別の二つのインターフェースからの送出により発生したフレームの衝突で，64 バイトを超えてしまった場合のパケットの廃棄にも，そのフレームが正常であるかどうかの確認のために FCS を利用します．

インターフェースの送信側において，フレームの送出中において，衝突を検出した場合，フレームの送出を停止するということは先に述べましたが，この停止の際に，ジャム信号として（通常 32 ビットの）ランダム・データを送出します．こうすると，64 バイトを超えた状態で発生した衝突も，受信側において FCS エラーによるフレーム廃棄が行われるので，誤った情報を

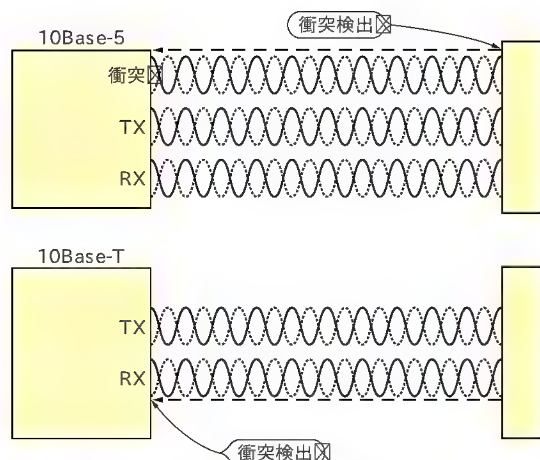


図8　10Base-5と10Base-Tの信号の違い

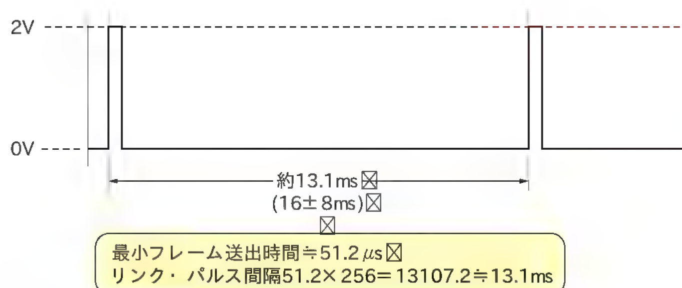


図9　リンク・パルス送出例

伝送するといった状況を回避することが可能となります．

このように，半二重通信であればFCSエラーが検出されることもありえますが，これが全二重通信になった場合は衝突は起こらないので，ほとんどのフレームはFCSエラー・チェックを通り抜けることになるでしょう．

● 宛て先確認

FCSエラーもない，廃棄対象とならない64バイトから1518バイトまでのフレームについては，送信先アドレスを確認することになります．先頭から6バイトを，自分自身のインターフェースに与えられているMACアドレスと比較し，同一であれば自分自身宛てとみなし，次の処理に移ります．

なお，ここで送信先アドレスの内容が，自インターフェースを示していなくても，ブロードキャストもしくはマルチキャストであり，フレームの送信を希望する先に自インターフェースが含まれるケースも次の処理に移る対象となります．

また，上位に位置するアプリケーションがパケットのキャプチャを目的としたものである場合には，自インターフェースと異なるほかのインターフェース間の通信も表示する必要があるため，上位からの設定によっては，自インターフェース以外であり，かつブロードキャスト，マルチキャストでない送信先を示すフレームについても，次の処理に移行させる場合があります．

● 受信バッファへの格納と上位への通知

こうして処理されたフレームは，プリアンブルとFCSを除去された後，上位にデータを渡すためにシステム・バスに接続された受信バッファに格納されます．また格納しただけでは上位は認識できないので，受信したフレームの大きさなどの追加情報も含めて格納した後，受信ステータス・レジスタの状態を変化させたり，システムに割り込みを発生させるなどします．

5　10Base-5と10Base-Tの違い

本章の最後に，10Base-5と10Base-Tの違いを見ます．ほとんどの動作において，10Base-5と10Base-Tは同じようにふるまいます．しかし，大きく二つの異なる動作があります．一つは衝突の検出に関わる部分であり，もう一つはリンク状態の確認です．

実は10Base-Tは，イメージとして10Base-5のトランシーバ・ケーブルから衝突信号を除いて，D-Subコネクタによる接続をUTPケーブルに置き換えたものとみなせます（規格上は異なるが）．

衝突の検出に関しては，10Base-5は電位差によってタップ・トランシーバで検出し，システムへは「衝突を通知するための信号線」で接続します．つまりタップ・トランシーバと接続されるD-Subケーブル上には送信ペアと受信ペアのほかに，衝突検出ペアがありますが，10Base-Tでは，送信ペアと受信ペアだけで，衝突検出ペアがありません．このため，衝突検出は受信ペア上にある情報を用いて確認を行うようになっています（図8）．

もう一つは，ネットワークの構成に関係するのですが，10Base-5は「バス型」ですが，10Base-Tはスター型で，PCなどに置かれたインターフェースから接続される対象は，ハブなどと1対1での接続になります．10Base-5のタップ・トランシーバはケーブルを介して電力を供給されるので，ケーブル接続がされない状態であれば動作しませんが，ハブは個別に電力が供給されているので，未使用ポートでも動作させることは可能です．

たとえば，8ポート・ハブに3台のPCを接続した状態で使用しているとして，1台がフレームの送出状態にあるとき，後の2台のインターフェースに対してハブからフレームが中継されるのは当然として，未使用の五つのポートにも送出するかどうかが問題となります．結論からいえば送出しません．10Base-Tのインターフェースは，ケーブルの先に，相手先の装置が接続されているかを確認するために，お互いに一定時間間隔で確認用のパルス（リンク・パルス）を送出しています．リンク・パルスの送出間隔は16±8msとされています（図9）．

Windowsなどを使用しているときに，Ethernetインターフェースの UTPケーブルを抜くと，たとえ通信を行っていない状態であっても警告が表示されるのは，このパルスによってリンク状態が確認されているからなのです．

まつもと・のぶゆき　（株）タムラ製作所

column 2

10Base-Tのレイヤ構成について

● OSI 7レイヤ

通信の世界でよく聞く表現にOSI 7レイヤ（階層）があります．これは通信を行うために必要となる処理について役割分担を決め，目的に合った手段を選択しやすくなるなどのために用いられます．

Ethernetの規格であるIEEE802.3では，OSI 7階層のうち下から二つのレイヤ1（物理層）とレイヤ2（データ・リンク層）が範囲となります．正確にはレイヤ2のデータ・リンク層はLLC副層（LLC Sub Layer）とMAC副層（MAC Sub Layer）で構成されますが，IEEEではLLC副層はIEEE802.2なので，IEEE802.3とIEEE802.2でレイヤ1とレイヤ2を決めていることになります．もっとも，最近ではLLC副層はあまり用いられていないので，事実上はIEEE802.3でレイヤ1とレイヤ2となっています（図A）．

● Ethernetのレイヤ1

レイヤ2にLLC副層とMAC副層があるように，レイヤ1にも多くの副層が存在します．そしてこれは用いるインターフェースの種類によって副層の数そのものが変化します．Ethernetでもインターフェースの速度によって副層の数は変化します（同じ機能で名称が違うだけだったり，位置が変わったために名称も変化したが目的は同じ物だったりと混乱しているようにも見受けられるが）．

10Mbpsである10Base-Tはけっこう微妙な位置にあります．10Mbpsのインターフェースとしては，ほかに10Base-2/5があります．しかし，10Base-Tと同じコネクタを使い，さらにはオート・ネゴシエーションで10Mbpsから1000Mbpsまで自動切り替えも可能な10/100/1000Base-TXがあります．10Base-Tはこの両方で動作するようになっています．つまり，オート・ネゴシエーションの10/100/1000Base-TXのインターフェースにも接続可能ですし，メディア・コンバータを使ってコネクタ形状を変えれば，10Base-5のネットワークでも使用可能となります（図B）．

とりあえず，レイヤ1の構成を10Base-2/5/Tで使用する場合，副層の数は四つになります．レイヤ2のMAC副層の下にくるものからPLS（Physical Layer Signaling），AUI（Attachment Unit Interface），PMA（Physical Medium Attachment），MDI（Medium Dependent Interface）となり，その先にPHYとしてUTPケーブルが接続されます．こうして書くと大ごとのようにも見えますが，実際はどれも大したことはありません．

PLSは，'0'/'1'の情報をマンチェスタ符号に変換する部分です．PMAはLSI内部に位置する差動ドライバとLSIの外にあるパルス・トランスを合わせたものです．MDIは単なるRJ-45コネクタのことですし，AUIにいたっては基板またはLSI内部の配線でしかないのです．

このAUIは，かつての10Base-5のトランシーバ・ケーブルに相当します．というのも10Base-5では，ネットワークは同軸ケーブルであったため，PHYに対して同一の場所にMAU（Medium Attachment Unit＝PMA+MDI）が存在していました．このためインターフェース・カードとMAUを接続する機能が必須であり，ラッチ付きD-Subケーブルとしてトランシーバ・ケーブルが使用されていました．

筆者の私見になりますが，10Base-Tに変わる際のイメージからすると，「MAU+PHY＝HUB」として，「AUI＝UTPケーブル」としてくれたほうが，よほどわかりやすいものです．

10Base-5と10Base-Tを比較すると，10Base-5でいう同軸ケーブルのところが，10Base-TではUTPケーブルになっています．このためMAU機能であるタップ・トランシーバはインターフェース・カードの中に内蔵されてしまったので，MAUと接続を行うケーブルは基板上の配線になってしまいました．

● Ethernetのレイヤ2

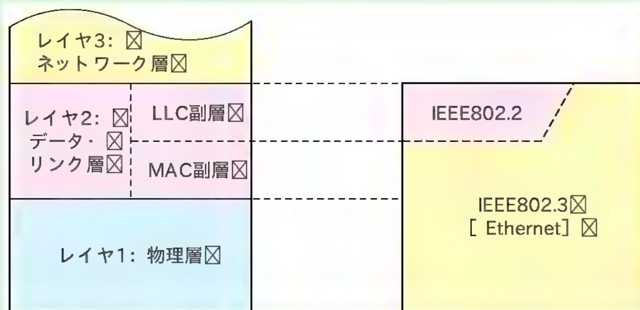
IEEE802.3（Ethernet）におけるレイヤ1は符号変換部以降です．つまり10Base-Tであればマンチェスタ符号にするところからしかありません．つまりEthernetとしてもっとも特徴的なCSMA/CDのメカニズムはレイヤ2が担当します．

EthernetはOSI 7レイヤによる表現と相性が悪いとよくいわれますが，その最大の要因がCSMA/CDでしょう．

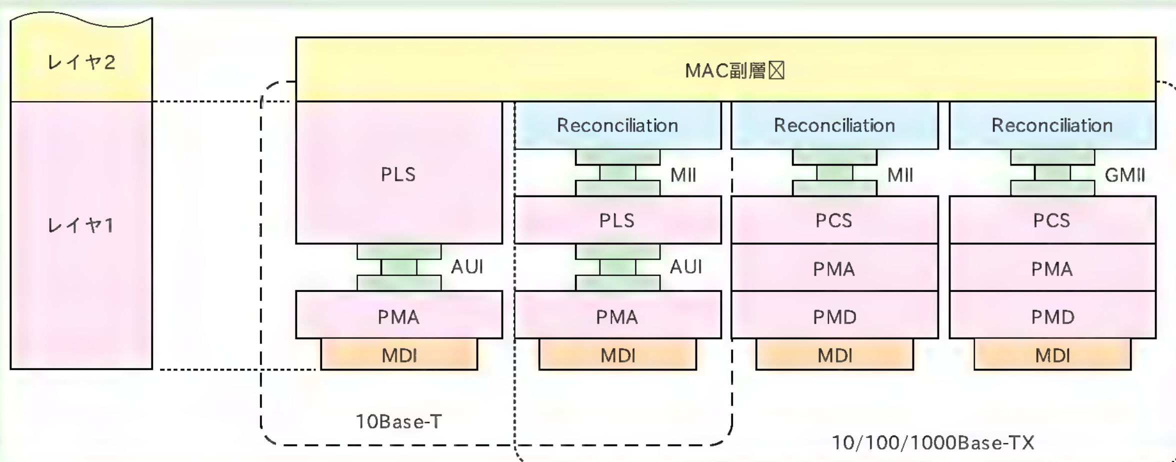
Ethernetのフレームを送出しようとした場合，ネットワークの空きを見て，空いていればフレームの送信を開始します．つまりフレームの処理をレイヤ2からレイヤ1に引き渡しているのです．しかし，インターフェースからみて外部であるネットワーク上において衝突が発生した場合，衝突検出の通知によって「レイヤ2からレイヤ1への引き渡しが中断される」のです．OSIで規定されている各レイヤには作業分担がありますが，EthernetのCSMA/CD動作においてはレイヤ2とレイヤ1の作業が同時に進行してしまいます．フレームの立場から見ると，ネットワークの状況によって，レイヤ2とレイヤ1の間を行ったり来たりしなければならないということです（図C）．

Ethernetのエンジニアで，「CSMA/CDで発生する衝突はメカニズムとして規定されているものであり，障害ではない」と発言する人を見かけます．いわんとするところはわかりますが，これではミスリードする可能性があると思っています．障害ではなくても支障があると思います．実際，伝送クロックの周波数に対して著しく低い値（通常，2割程度）の情報転送能力しかもてないわけですし，音声や映像のようなストリーム系情報を伝送しようとすると品質に影響を与えます．

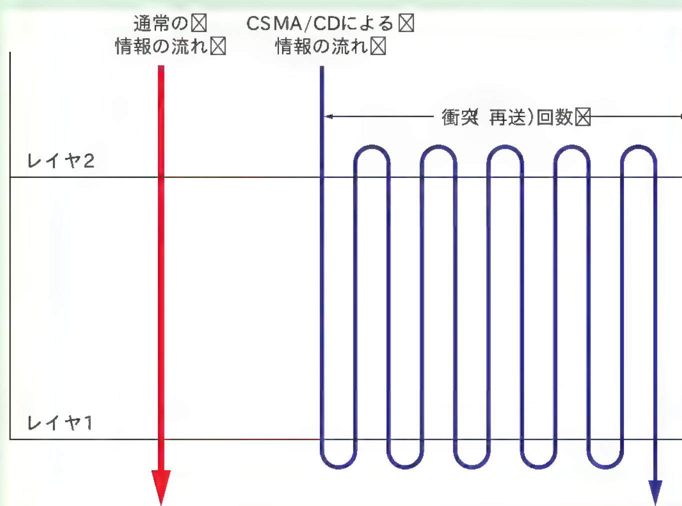
しかし，ここでちょっと考えかたを変えてみましょう．まずTCP/IPプロトコル・スタックにおけるレイヤ4の動作を見てみます．レイヤ4の中でもUDPではなくTCPのほうです．TCPは再送機能をもっています．送信したパケットが，なんらかの理由で正常に相手が受信できなかったとき，同じパケットを再度送出することにより正確な通信を実現しています．再送機能があるといっても，一つのパケットを送



図A　レイヤ1と2，および3以上



図B　インターフェース別のレイヤ1の構成



図C　フレーム衝突時のレイヤ1と2のやり取り

出した後，正常に到着したかを確認して，次のパケットを送出するようなことはせず，ウインドウ・サイズによって規定された範囲のパケットを連続送出して，問題があれば再送を行うのです．

ここで話をCSMA/CDに戻します．キーワードはウインドウ・サイズです．CSMA/CDの動作を考えるとき，最初に送る情報をビット単位で考えます．第1章で Ethernetでは，相手先にフレームを送出するというより，ネットワークに置く」と表現しています．その考えかたで，Ethernetインターフェースは，ビット単位でフレームの一部をネットワーク上に置いていきます．置いたビット情報が正常であれば次の情報をおくのですが，問題が発生（この場合，衝突の発生）すると通知があり，最初からやり直します．

以上の動作を対称づけると（まったく一致するわけではないが），CSMA/CDでいうキャリア・センスをTCPのセッション確立に，衝突の検出をACK信号未着に，ビット情報をパケット情報に，フレームをファイルに，フレームの大きさをウインドウ・サイズに置き換えてみると，やっていることは同じことなのです．

ファイルから見て，TCPで再送を行っていることは故障ではありません．20パケットで済むはずのファイルの転送に30パケットを使ったとしても，それは問題ではないのです．

先ほど，CSMA/CDが，映像や音声のようなストリーム系情報に向かないと書きました．これはTCPも同じです．インターネット電話でTCPを使おうとすると，品質に悪影響を与えることがあり，一部の例外を除いて使用されません．このような場合，TCPではなくUDPが使用されます．つまりEthernetでも音声情報や映像情報を用いる場合であれば，CSMA/CDを用いる半二重ではなく，全二重を選択すればよいのです．

全二重を選択したEthernetインターフェースは，上位であるレイヤ3から到着するパケットにレイヤ2ヘッダを付与するだけなので，ほかの通信で用いられるレイヤ2と何ら変わるものではありません．違いはCSMA/CDだけですが，これも，細部の違いこそあれ，ほかのレイヤで行われている処理と同じようなことがなされています．

第3章



FPGAでオリジナル仕様LANカードを作る

10Base-T対応 LANカードの設計/製作

松本 信幸/山武 一朗

10Base-Tについて理解したところで，本章では10Base-Tに対応したEthernetコントローラをFPGAで実現する．まずハードウェアの構成について考察したあと，ブロックごとに各部の動作を解説する．本章で設計するEthernetコントローラはVHDLで記述してあり，フリー版の設計ツールで，論理合成から配置配線まで実行できる．

（編集部）

1 ハードウェア構成の検討

● Ethernetは差動信号を使う

Ethernetで用いる信号は差動信号です．差動信号は2本の伝送路をペアで使用します．差動信号を用いる利点は二つあり，一つは情報を伝送するために必要な電位を決定する基準（通常グラウンド・レベル）を接続単位に決定し，基板などで用いるグラウンドと異なった電位を使うことができるという点があります．

もう一つはTTLなどで用いられるような，基準電位に対する信号の電位で情報を伝送するのではなく，二つの信号の状態の違いにより情報を伝送する点にあります．差動信号をAとBとすると，TTLでいう‘1’の状態はA=‘1’，B=‘0’で，TTLでいう‘0’の状態はA=‘0’，B=‘1’となります．このような伝送方法の利点は，外乱電磁波によって発生するノイズに強いため，筐体間が離れた機器どうしの通信に適しているということです．

● 物理層（PHY）デバイスを使わない方法はないか

一般的なPCIバスやIDEなどのインターフェースは，TTLもしくはLV-TTLの通常のロジック信号が使われています．このような場合はFPGAを直結できますが，Ethernetは差動信号を使っているので，FPGAなどのロジック・デバイスと直接には接続できません．そのため，一般的には，Ethernetにも対応するFPGA評価ボードなどでは，ロジックで構成できる論理層（MAC）部分はFPGAで実現するものの，差動信号を直接扱うPHY部分には，専用のPHYデバイスを実装しています．

しかし，今回の特集の目的の一つは「Ethernetの動作を物理レベルから体験して理解すること」にあります．その観点からすると，たとえPHYデバイスといえど，できあいのデバイスは使いたくないところです．

最近のFPGAは高機能化がすすみ，扱える信号はTTLやLV-TTLだけでなく，差動系の信号も直接接続できるようになってきています．そこで差動信号を扱う部分は，FPGAのもつ差動ドライバ/レシーバ機能を使って実現できないかどうかを考えます．

● FPGA評価キット

今回使用するFPGA評価ボードは，アルテラ製FPGA EP1S10-FBGA780を搭載したStratix評価キットです．このボードを，インターフェースを実装するターゲットPCのPCIバスに接続し，ターゲットPCからPCIバスを介してパケットの送受信を行います．

Stratixデバイスには当然ながら差動信号にも対応しています．その中から，電圧的に比較的自由度のある「3.3V PCML」というモードを使うことにします．ただし，今回はコラム1のような問題から，送信部にはLV-TTLを使って，図1のような疑似的な差動回路を構成します．

今回はあくまでもEthernetの学習用という部分にこだわり，可能な限りFPGAだけで構成することを考えました．よって今回の回路が電気的にEthernetの規格を満足しているかというと，それはまったく保証できません．しかし，一般的な数mのLANケーブルを使って接続する分には，問題なく通信できます．

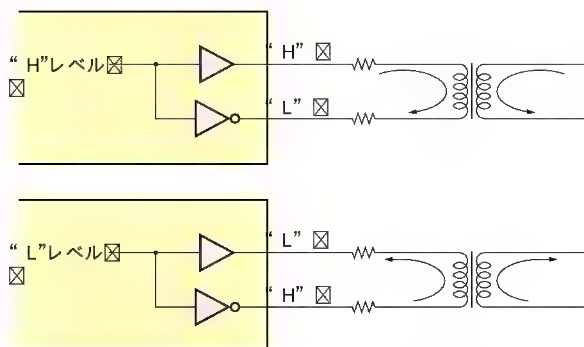


図1 ロジック信号を疑似的に差動信号にして送信

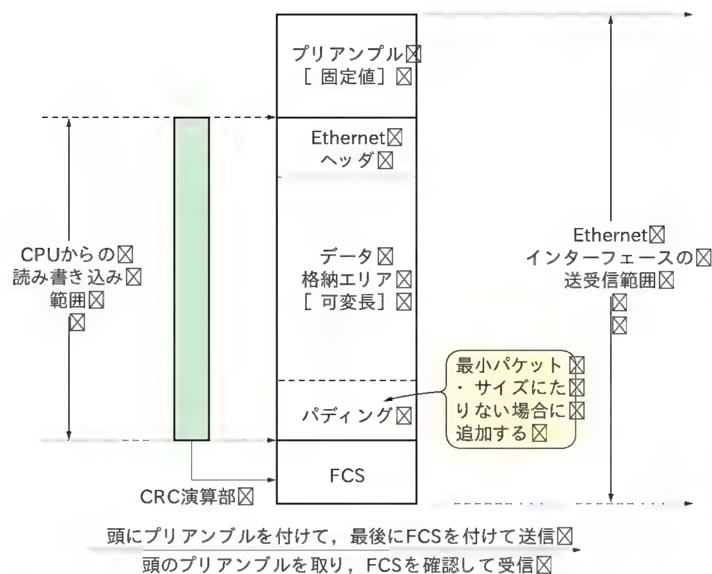


図2　送受信バッファ構成

● Ethernetコントローラの構成

次にEthernetコントローラの内部構成について考えます．

回路を簡単にするなら，コントローラ内のバッファをほとんどなくし，FCSはおろか先頭のプリアンブルまで含んだEthernetフレームをソフトウェア(ドライバなど)で作成し，コントローラとしては1バイトのパラレル→シリアル変換や信号レベル変換，そしてマンチェスタ符号への変換を行うだけにしてしまうという手段もあります．

フレームの送信を開始したら，最初の1バイト分の送信を終えるうちに次のバイトを書き込み，これをFCSの最後まで遅れのないよう繰り返して送信を行うわけです．このデータ転送をCPUが行うとなると，かなりの負荷になります．実質的にフレーム送信中はほかの処理はできなくなるでしょう．

そこで，送信するフレームのデータをあらかじめ用意しておき，送信開始を指示したら，あとはCPUの力を借りずにハードウェアが自動的にフレームを送信するという構成がよいでしょう．プリアンブルは送信するデータがどんなものでも同じパターンなので，これはハードウェアが自動的に生成すればよく，フレームの最後のFCSもハードウェアが自分で計算して最後に付加すれば，CPUの負荷が減り，パケット送信処理のプログラムも楽になります．

受信の場合も同様でハードウェア内にバッファを用意し，プリアンブルはタイミング検出用に受信するだけでバッファには格納せず，実際のデータ部分が始まったらそこから受信したデータを格納していきます．またフレーム最後のFCSを受け取ったら，それまで受信したデータから計算したCRCと一致するかどうかをハードウェアで判定し，受信したフレームが正しいかどうかまでをCPUに知らせれば，CPUのパケット受信処理も楽になります(図2)．

column 1

Stratixデバイスでの差動信号

10Base-Tでは，送信アイドル時は差動信号の電圧差が±50mVという規定があります．しかしStratixデバイスの差動モードでは，片方が"H"のときはもう片方が"L"になり，両方とも"H"が"L"，またはハイ・インピーダンス(Hi-Z)という状態は取れないようです．つまり送信アイドル状態を作れないのです．

実は当初，送信側も3.3V PCMLモードを採用し，送信アイドル時も電位差のある状態のままでした．これでも10Mbps専用のポートとは通信ができるのですが，10M/100Mbpsの両方に対応したハブやLANカードとはうまく通信が行えませんでした．10M/100Mbpsの両方に対応したポートは，オート・ネゴシエーションといって通信速度やモードを自動判定する動作がありますが，送信アイドル時の電圧差が大きいと，このときの動作で問題を起すようです．

ちなみに差動入力のほうは，信号間に電圧差がない状態は"L"とみなしているようなので，10Mbps専用の場合は，これはこれで問題ありません．

● PCIバス対応のEthernetカード

今回使用するStratix評価キットはPCIバスに対応しています．そこでPC/AT互換機にPCIバスで接続する形状の，10Base-T対応LANカードとします．

PCIバスではCPUに負荷をかけずにデータ転送を高速に行うバス・マスタ転送方式がありますが，今回はハードウェアをシンプルにするという意味から，バス・マスタ方式は採用しません．バス・マスタ方式を採用しないため，データ転送はCPUが行う必要があります．

今回は送信バッファを用意するので，1フレーム分のデータ転送が間に合わなくなることはなく，またCPUがほかの仕事で忙しくても，次のフレームの送信が遅れるだけで，全体としてパフォーマンスが悪くなることはあっても，エラーは発生しないでしょう．しかし，受信に関しては，ほかのネットワーク機器から次々とフレームが送られてくる可能性があります．あるフレームが受信バッファに溜まった状態でCPUがそれを取り出す前に次のフレームが来てしまうと，データを格納しておく場所がなくなってしまい，受信バッファ・オーバフロー・エラーが発生してしまいます．

そこで，CPUが多少忙しくてもフレームを取りこぼさないように，受信バッファは複数個用意しておくことにします．

2 FPGA評価キットと拡張基板

図3にFPGA評価キットと開発マシン，ターゲット・マシ

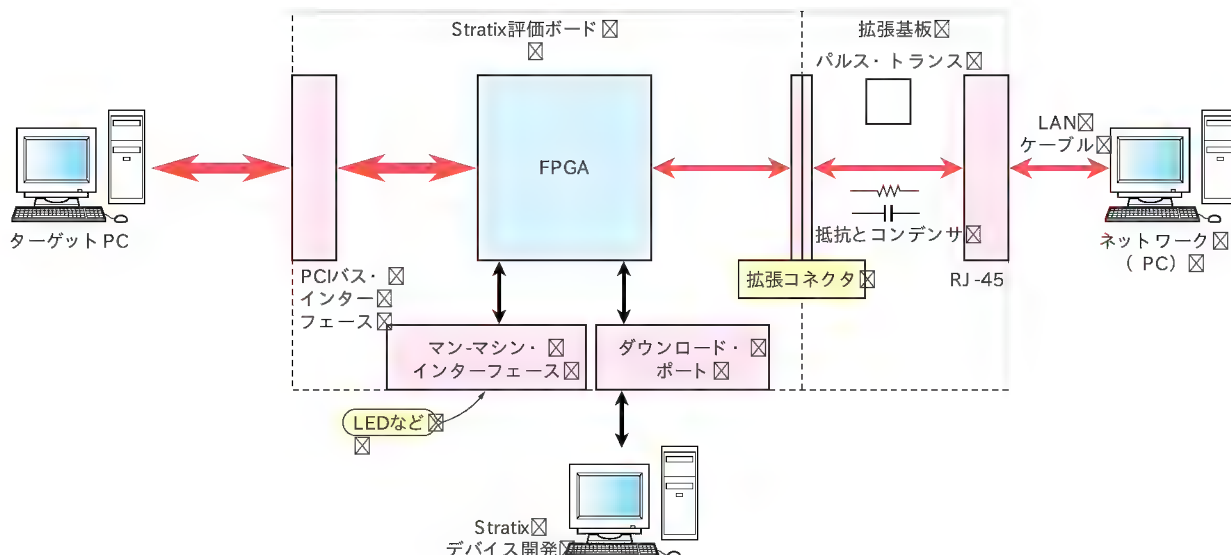


図3　システム構成図

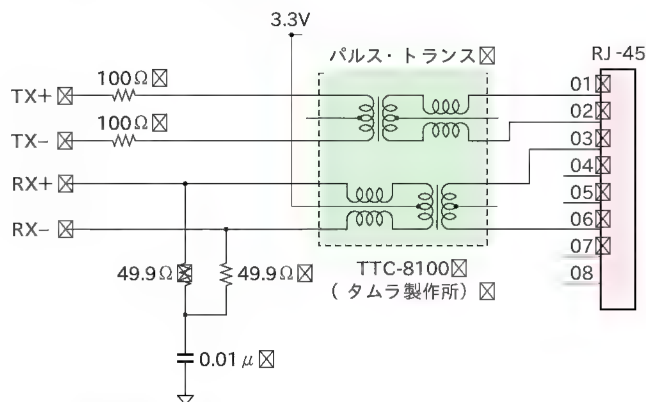


図4　拡張基板の回路図

ン，ネットワークとの接続関係を示します．

● PCIバス・インターフェース

今回対応を考えているPCIバスは，バス幅32ビットでクロック33MHzの仕様を考えます．なお特集の趣旨から離れるので，PCIバスについての詳細に関しては割愛します．

ちなみにEP1S10-FBGA780は，5V系PCIバスには対応していません．Stratix評価キットを見ると，PCIカード・エッジとFPGAの間には，特にアイソレーション系のバッファなどは見当たらないので，本来であればこのボードを5V系PCIスロットで使うことには問題があります．しかし，現実的に，現在一般的に使われているPCでは，5Vトレラントな3.3V系チップ・セットが使われており，5V系のPCIスロットとはいえ，実質的には3.3Vの信号になっているので，問題はないでしょう．

● 10Base-Tインターフェース

今回実現するのは10Base-Tなので，10Base-Tの標準インターフェースである8ピンのRJ-45コネクタを使用します．ピンの仕様については標準の1-2番ピンを送信信号ペア，3-6番ピンを受信信号ペアとして，MDIの仕様で使います．

また今回は，IEEE802.3afのPower over Ethernetは使用しないので，4，5，7，8番ピンは空きピンです．空きピン処理はオープン(未接続)でかまいません．なお，今回は10Base-Tに限定するので，オート・ネゴシエーションなどは行いません．

● 拡張基板インターフェース

Stratix評価キットは標準ではRJ-45コネクタが実装されていないので，Stratix評価キットの拡張コネクタを使って，拡張基板を接続します．

図4に拡張基板の回路図を示します．拡張基板上には，基板上とネットワーク接続した相手機器との電流のループをカットするためのパルス・トランスを実装します．パルス・トランスとRJ-45コネクタは図のように直結するだけです．

パルス・トランスとFPGAの間には，多少の抵抗とコンデンサを使います．パルス・トランスにはTTC-8100(タムラ製作所)を使いました．

まず送信側ですが，今回はFPGAの出力ピンをLV-TTLモードにし，たがいに状態を反転した値を出力することで，疑似的に差動信号出力を実現します．ダンピング抵抗は60Ω～100Ω程度でよいでしょう．FPGA側のピンの動作モードとして，LV-TTLを指定することを忘れないでください(デフォルトでLV-TTLになっているが)．

受信側はFPGAも差動入力にして使います．RJ-45コネクタの3-6番ピンから，送信側と同じようにパルス・トランスに接続し，パルス・トランスの出力をFPGAの差動入力ピンに接続します．また10Base-Tにおける終端は100Ωを基本としているので，受信側のペア端子間に100Ωの抵抗を用いて終端しておきます．ただし，ここは本当に100Ωを接続するだけでも良いのですが，コネクタ接続を行うことを考慮し，安定性を高め

るためにコンデンサを使います．コンデンサを入れる場所は，抵抗の中間点とするため，終端抵抗を二つに分け，その中間点とグラウンドの間にコンデンサを入れます．なお，50Ωの抵抗はないので49.9Ωを使います．

● FPGAピン配置

PCIバス・インターフェースなどは，Stratix評価キットの基板上ですでに配線されているので，取扱説明書などの資料のとおりに定義します．送受信の基準タイミングとなるクロックは，基板上にクロックが実装されているので，これも取扱説明書などに指示のあるピンから入力します．

ピン配置をユーザで決められるのは，拡張基板との接続部分となります．とはいえ，Stratix評価キットであらかじめ拡張コネクタに配線されたピンの中から選択しなければなりません．

通常よく使うLV-TTLの信号なら，ほとんどのピンで自由に割り当てることができます．しかし，今回は受信側は差動インターフェースとして3.3V PCMLモードを使用するため，Stratixデバイスでは，差動モードで使用できる送受信のピンの組み合わせに制限があります．よって，Stratix評価キットの拡張コネクタに配線されているピンの中で，差動モードでも使えるピンである必要があります．

ちなみに，Stratix評価キットではLVDSの評価用として，CN6とCN7にLVDSモードで使えるピンがあらかじめ配線されています．しかし取扱説明書によると，LVDSの受信端子には終端抵抗としての100Ωがあらかじめ実装されているとあります．この終端抵抗は今回の回路では不要なので，CN6に配置された信号は使えません．

そこで，CN3に配線されている信号の中から，3.3V PCMLで使える組み合わせを考慮して，表1のようなピン配置にしました．

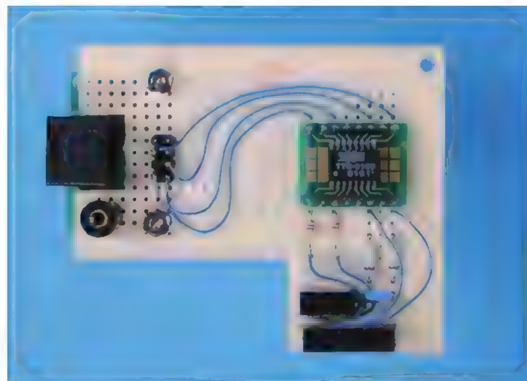
Ethernetの送受信で必要となるクロックは，Stratix評価キットのオンボードに実装されているクロックを使います．ただし，コラム2で説明しているPLL機能の制限で，特定のクロックの組み合わせでは，クロック入力ピンが限定されることがあるようです．今回は表1に示すようにAC17番ピンを使用しました．

製作した拡張基板と，拡張基板を実装したときのStratix評価キットの概観を写真1に示します．写真を見るとわかるように，Stratix評価キットをPCIスロットに実装したとき1スロット分で済むように，拡張基板は裏返しで接続するようにしてスリム化しました．

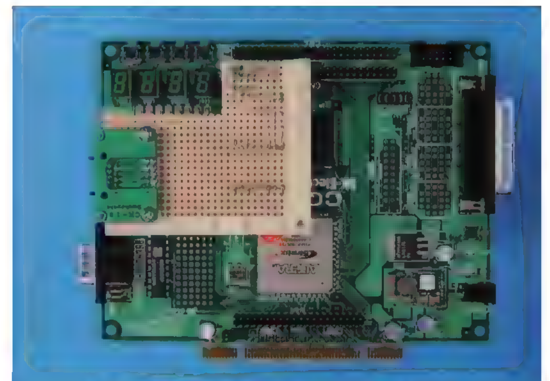
3 Ethernetコントローラの設計

● 全体の構成

今回はシステム・バスとしてPCIに接続することを考えます



（a）拡張基板



（b）拡張基板を実装したStratix評価キット

写真1 製作した拡張基板概観

表1 FPGAピン配置（Ethernet送受信信号のみ）

信号名	ピン番号	動作モード
CLK33	AC17	LV-TTL
TXp	K28	LV-TTL
TXn	U24	LV-TTL
RXp	J27	3.3V PCML
RXn	J28	3.3V PCML

注：設計ツール上は，RXpの信号のみ'RX'として定義する．差動で対となる信号は自動的に決定される．

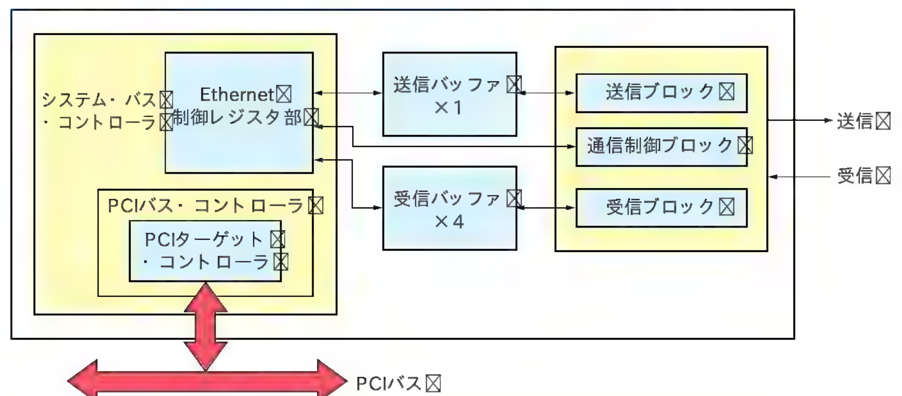


図5 Ethernetコントローラの全体構造

表2 VHDLソース・ファイルの内容

DualRAM.VHD DualRAM.???	Stratix用デュアル・ポートRAM定義ファイル(VHDL) Stratix用デュアル・ポートRAM定義関連ファイル
X3PLL.VHD X3PLL.???	Stratix内蔵PLL制御用定義ファイル(VHDL) Stratix内蔵PLL制御用定義関連ファイル
LED_CTRL.VHD	LED点灯制御VHDLファイル 人間が目で見てわかるように，消灯タイミングを約0.5秒引き延ばす処理を入れたもの
ROM_ACC.VHD	シリアルROMアクセス・コントローラ(ダミー) MACアドレスを格納しているシリアルROMのアクセス・コントローラ． 今回は実際には外部のシリアルROMを読まずに，内部的にダミー・データを返している． MACアドレスはオフセット＋10hからの6バイトに,「AC DE 48 00 00 01」を格納している． また書き込みテスト用に，オフセット＋80hから4バイトが，R/W可能レジスタを実装している
SYNC_a.VHD SYNC_b.VHD	非同期信号同期化処理(TypeA) 非同期信号同期化処理(TypeB) 非同期なクロックで動作する回路でフラグをやりとりする場合に信号の受け渡しが正しく行われるようにするモジュール．TypeAは信号をハンドシェーク式にやりとりする場合(複数クロックにわたりセットされる)信号用． TypeBは1クロック期間のみセットされる信号の場合に使用する
PCI_CTRL.VHD PCI_TGT.VHD	PCIバス制御用 PCIバス・ターゲット・シーケンサ制御用 この二つのファイルでPCIバス・プロトコルを制御する
SYSBUSC.VHD	システム・バス接続モジュール Ethernetコントローラの各種制御レジスタと，送受信バッファをPCIバスに接続するモジュール．PCIバス側からみた各種メモリ・マップはこの中で定義される
EtherC.VHD	Ethernetコントローラ・モジュール 今回の設計のメイン部分にあたる
CRC32.VHD	CRC32計算用モジュール
PCI_NIC.VHD	PCIバス対応Ethernetカード・トップ・モジュール以上の各モジュールを束ねる，設計データのトップ・モジュール

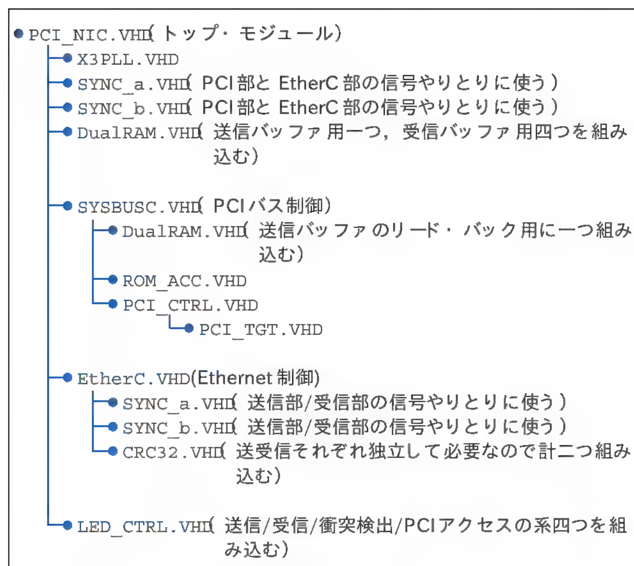


図6 VHDLソース・ファイルと階層構造

が，PCIの仕様にべったりなメモリ・マップや構造にはせずに，Ethernetコントロール部とシステム・バス制御部とを分離して，設計データをほかのバスに接続しやすいように考えます．設計したEthernetコントローラ全体の構成を図5(p.59)に，VHDLソース・ファイルの内容を表2に，VHDLソース・ファイルの階層構造を図6に示します．

PCIバス側から見たメモリ・マップやレジスタ構成を表3(pp.62-63)に示します．システム・バスへの接続部分などについての詳細は，誌面のつごうで省略します．ここでは特集の趣旨でもあるEthernetコントロール部について詳しく解説します．

図7にEthernetコントロール部内のブロックの概要を示し

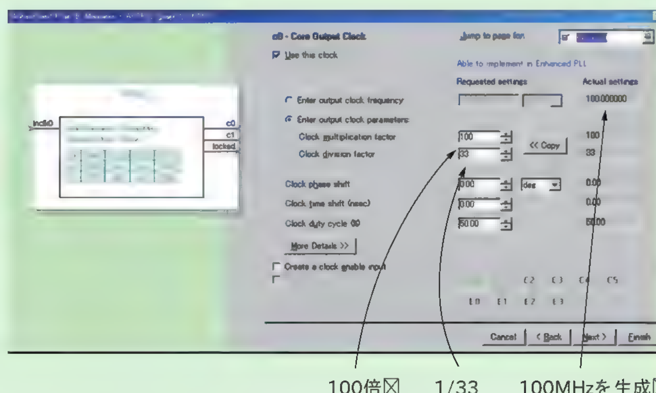
column 2

Stratix評価キットのクロック

Stratix評価キットの取扱説明書では，実装されているクロックは33.333MHzと書かれていますが，筆者が実際に使ったボードには33.000MHzが実装されていました．このあたりはロットによって異なっている可能性もあるので，実際に使用する場合には実装されているクロックを確認する必要があるでしょう．

今回のEthernetの制御では，後述するように送信部に20MHz，受信部に100MHzのクロックが必要です．そこでStratixデバイスに内蔵されているPLL機能を活用し，評価ボードに実装されているクロックから20MHzと100MHzのクロックを生成してみます．

たとえば33.000MHzが実装されていた場合は，20倍してから1/33に分周することで20MHzを，100倍してから1/33に分周することで100MHzを生成できます(図A)．



図A MegaWizardによるPLLの逓倍/分周設定画面

ます．Ethernetコントロール部は大きく分けて送信ブロック，受信ブロック，通信制御ブロック，リンク制御ブロックに分けられます．以降でそれぞれの動作概要について説明します．

● 送信ブロックの構成

送信バッファに書き込まれたデータを送信するためのブロックで，入力情報として送信データ，送信バイト数，送信開始信号などがあり，出力情報としてマンチェスタに符号化されたシリアル出力，送信動作フラグ，送信完了通知などがあります．

送信データは，PCIバスに接続された送信バッファから実際に送信するデータを取り出す部分です．送信開始信号と送信バイト数は，PCIバス側に用意した送信開始レジスタや送信バイト数レジスタの信号が伝わってきます．

送信許可信号は通信制御ブロックからの情報で，ネットワークに空きがあり，送信可能である場合にイネーブルとなります．送信動作フラグは，通信制御ブロックからの送信許可を受けて，送信ブロックがフレームの送信動作を行っているときにイネーブルになる情報です．送信動作が終了すると取り消されます．

シリアル出力は，マンチェスタ符号に変換した状態で実際にネットワークに送り出す信号です．

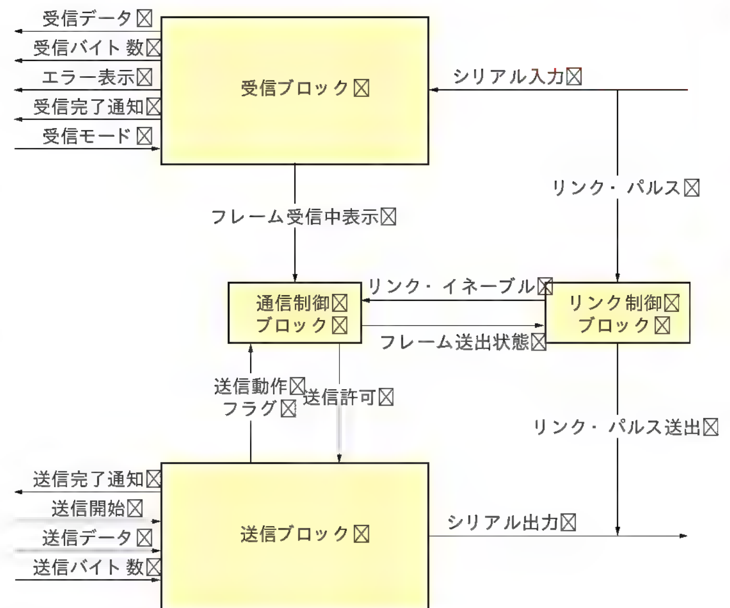


図7　Ethernetコントロール部の各ブロックの概要

● 受信ブロックの構成

こちらは送信ブロックとは逆で，ネットワークから自分宛てのフレームなど必要なものを選択し，受信バッファに対して出力するためのブロックです．ブロックに対する入力情報としてマンチェスタ符号のシリアル入力があり，出力情報としてフレーム受信中表示とエラー表示，受信データ，受信バイト数，受信モード，受信完了通知などがあります．

シリアル入力は，マンチェスタ符号による差動信号で，拡張基板を介してネットワークから入力されます．

フレーム受信中表示は，シリアル入力においてオルタネート・データが検出されるとアクティブになる信号です．

受信データは，受信バッファに接続されたインターフェースで，受信したデータをバイト単位に書き込んでいきます．受信バイト数はPCI側の受信バイト数レジスタに反映されます．

エラー表示は，受信したフレームが自分宛てであったにもかかわらず，CRCエラーなどにより正常ではなかった場合にアクティブになる信号です．エラーの種類は，フレーム・エラー（短すぎ/長すぎ）とCRCエラーです．

受信モードは，受信しようとするフレームについて，自分宛ての物だけを取り込むのか，エラーが発生していないすべてのフレームを取り込むのかなどを識別する情報で，PCIバス側の各種制御レジスタからの指示によって決定します．

受信完了通知はフレームの受信が完了して，受信バッファに有効なデータが存在する場合にアクティブになります．

● リンク制御ブロックの構成

ケーブルを介して，相手のネットワーク機器が接続されているかどうかを確認するためのブロックです．出力信号としてリンク・パルス送出，リンク・イネーブルがあり，入力情報としてはリンク・パルス，フレーム送信状態があります．

リンク・パルス送出は，相手のネットワーク機器に対して，みずからのインターフェースが動作可能な状態にあることを定期的に通知するためのものです．リンク・パルス信号送出の周期は，規定では16±8ms間隔ということになっているので，今回は13msで行うようにしています．

フレーム送信状態は，通信制御ブロックから入力される情報で，文字どおりフレームの送信動作を行っていることを示す信号です．この信号は，送信ブロックから通信制御ブロックに対して出される送信動作フラグ信号とは異なり，実際にフレームの送信動作を行っているときにのみアクティブとなります．その目的は，インターフェースのリンク確認は周期的に行うことになっているのですが，その周期とフレームの送出が同時になった場合に，リンク・パルスであるリンク・パルス送出によって送信中のフレームに悪影響を与えないようにしなければならないためです．

リンク・パルスは，受信ブロックのシリアル入力を共用します．リンク制御ブロック内にカウンタ・タイマを用意し，リンク・パルスとしてシリアル入力より何らかの信号が入力された場合，相手のネットワーク機器が動作可能な状態にある（要は接続されている）とみなします．これは，対向となる相手側が送出してくるリンク・パルスなので，相手側が正常に動作している限り16±8msの間隔で信号パルスが到着することになります．

リンク・イネーブルは，リンク・パルスの入力によりクリアされるカウンタ・タイマの出力信号を用います．カウンタ・タイマがカウント動作しているときは，リンク・イネーブルはアクティブ状態となります．カウンタが満杯になると，カウン

表3　設計したEthernetコントローラのPCIバス側の各種レジスタ仕様

ベンダID	6809h
デバイスID	8010h
コマンド・レジスタ	メモリ空間ビット
ステータス・レジスタ	DEVSEL中速応答
基本クラス	02h(Ethernetコントローラ)
サブクラス	00h
プログラムI/F	00h
リビジョンID	01h
ベース・アドレス・レジスタ0	1Mバイト・メモリ空間要求
サブベンダID	6809h
サブシステムID	8010h
割り込み	INTA#使用

(a) PCIコンフィグレーション・レジスタ

オフセット	用　途
+0_0000h	Ethernet制御レジスタ空間
+1_0000	受信バッファ0空間(読み出し専用)
+1_0800h	受信バッファ1空間(読み出し専用)
+1_1000h	受信バッファ2空間(読み出し専用)
+1_1800h	受信バッファ3空間(読み出し専用)
+1_8000h	送信バッファ空間(読み書き可能)

(b) PCI空間メモリ・マップ

オフセット	リード/ライト	ビット	用　途
+000h	R	31～0	シグネチャ・レジスタ 68098010h
+008h			▶MACアドレス制御レジスタ
	W	31	MACアドレス設定レジスタ ライト・イネーブル・ビット
	R/W	18～16	MACアドレス設定レジスタ アドレス・レジスタ
	R/W	7～0	MACアドレス設定レジスタ データ・レジスタ
			(ライト・イネーブル・ビットに'1'を立てながらデータ・レジスタに書き込み)
+00Ch			▶シリアルROMアクセス制御レジスタ
	R	31	シリアルROMアクセス ビジー・ビット('1'でアクセス中)
	W	31	シリアルROMアクセス スタート・ビット('1'を書き込むとアクセス開始)
	R/W	30	シリアルROMアクセス 方向指定ビット('1'で読み出し/'0'で書き込み)
	R/W	23～16	シリアルROMアクセス アドレス・レジスタ
	R/W	15～8	シリアルROMアクセス ライト・データ・レジスタ(デバッグ用暫定)
	R	7～0	シリアルROMアクセス リード・データ・レジスタ
+010h			▶割り込みステータス・レジスタ
	R/WC	31	送信系エラー割り込みステータス('1'でエラー割り込み)
	R/WC	16	送信完了割り込みステータス('1'で送信完了割り込み要求)
	R/WC	15	受信系エラー割り込みステータス('1'でエラー割り込み)
	R/WC	3	受信バッファ3受信完了割り込みステータス('1'で受信完了割り込み要求)
	R/WC	2	受信バッファ2受信完了割り込みステータス('1'で受信完了割り込み要求)
	R/WC	1	受信バッファ1受信完了割り込みステータス('1'で受信完了割り込み要求)
	R/WC	0	受信バッファ0受信完了割り込みステータス('1'で受信完了割り込み要求)
+014h			▶割り込みマスク・レジスタ
	R/W	31	送信系エラー割り込みマスク('1'で割り込み許可)
	R/W	16	送信完了割り込みマスク('1'で割り込み許可)
	R/W	15	受信系エラー割り込みマスク('1'で割り込み許可)
	R/W	3	受信バッファ3受信完了割り込みマスク('1'で割り込み許可)
	R/W	2	受信バッファ2受信完了割り込みマスク('1'で割り込み許可)
	R/W	1	受信バッファ1受信完了割り込みマスク('1'で割り込み許可)
	R/W	0	受信バッファ0受信完了割り込みマスク('1'で割り込み許可)
			(PCIバス・リセット時はオール'0')
+020h			▶受信コンフィグレーション・レジスタ
	R/W	31	受信部リセット・レジスタ('1'でリセット/'0'でリセット解除)
			(PCIバス・リセット時'1')
	R/W	30	CRCエラー・パケットも受信バッファに取り込む('1'で取り込む)
	R/W	29	フレーム・エラー・パケットも受信バッファに取り込む('1'で取り込む)
+024h			▶受信ステータス・レジスタ
	R/WC	31	受信バッファ・オーバフロー('1'を書き込むとエラー・ステータス・クリア)
			(受信バッファ・オーバフローを起した場合は，一度受信部リセットをかけること)
	R/WC	30	CRCエラー・パケット検出('1'を書き込むとエラー・ステータス・クリア)
	R/WC	29	フレーム・エラー・パケット検出('1'を書き込むとエラー・ステータス・クリア)
+030h	R/WC	15～0	紛失パケット・カウンタ(書き込み動作をするとクリア)
+034h	R/WC	15～0	CRCエラー・パケット・カウンタ(書き込み動作をするとクリア)
+038h	R/WC	15～0	フレーム・エラー・パケット・カウンタ(書き込み動作をするとクリア)
+040h			▶受信バッファ0制御レジスタ
	R/WC	31	受信完了ステータス('1'で受信完了)
	R	30	CRCエラー・パケット受信(エラー・パケットを取り込むときのみ)
	R	29	フレーム・エラー・パケット受信(エラー・パケットを取り込むときのみ)
	R	10～0	受信バイト数

(c) Ethernet制御レジスタ空間レジスタ・マップ〔WC：ビットを立てた値を書き込むとクリア(ライト・クリア)〕

表3 設計したEthernetコントローラのPCIバス側の各種レジスタ仕様(つづき)

オフセット	リード/ライト	ビット	用途
			▶受信バッファ1制御レジスタ
+044h	R/WC	31	受信完了ステータス('1'で受信完了)
	R	30	CRCエラー・パケット受信(エラー・パケットを取り込むときのみ)
	R	29	フレーム・エラー・パケット受信(エラー・パケットを取り込むときのみ)
	R	10～0	受信バイト数
			▶受信バッファ2制御レジスタ
+048h	R/WC	31	受信完了ステータス('1'で受信完了)
	R	30	CRCエラー・パケット受信(エラー・パケットを取り込むときのみ)
	R	29	フレーム・エラー・パケット受信(エラー・パケットを取り込むときのみ)
	R	10～0	受信バイト数
			▶受信バッファ3制御レジスタ
+04Ch	R/WC	31	受信完了ステータス('1'で受信完了)
	R	30	CRCエラー・パケット受信(エラー・パケットを取り込むときのみ)
	R	29	フレーム・エラー・パケット受信(エラー・パケットを取り込むときのみ)
	R	10～0	受信バイト数
			▶送信コンフィグレーション・レジスタ
+060h	R/W	31	送信部リセット・レジスタ('1'でリセット/'0'でリセット解除)
			(PCIバスリセット時'1')
			▶送信ステータス・レジスタ
+064h	R/WC	0	衝突検出('1'で検出)
+06Ch	R/WC	15～0	衝突検出カウンタ(書き込み動作をするとクリア)
			▶送信バッファ0制御レジスタ
+070h	R	31	送信中('1'で送信中/'0'で送信完了＝送信バッファ空き)
	W	31	送信開始('1'で送信開始)
	R	30	送信エラー(衝突が激しくて送信できなかった場合に'1')
	R/W	10～0	送信バイト数

(c) Ethernet制御レジスタ空間レジスタ・マップ(つづき)

タ・タイマはカウントを停止して，リンク・イネーブルを取り下げます．カウント時間は16±8msなので，最大24msとします．

● FPGA内部通信制御ブロックの構成

このブロックが，CSMA/CD方式のコアとなる部分です．

出力信号としては，リンク制御ブロックに対してフレーム送信状態があり，送信ブロックに対して送信許可があります．入力信号には，リンク制御部からリンク・イネーブルが入力され，送信ブロックから送信動作フラグ，受信ブロックからフレーム受信中が入力されます．

このブロックにより，CSMA/CD動作の制御を行っています．逆にいえば，このブロックの動作をディセーブルし，リンク制御部からのリンク・イネーブルを直接送信ブロックの送信許可として，さらに送信ブロックからの送信動作フラグをリンク制御部のフレーム送信状態として用いることで全二重通信が可能になります．

4 Ethernetコントロール部の詳細

次はHDLのソースを示しながら，Ethernetコントロール部の詳細を解説します．

● シリアル出力制御と送信リンク・パルス出力部

リスト1にシリアル出力制御と送信リンク・パルス出力部を示します．今回の送信部はLV-TTL出力で疑似的に差動出力を

リスト1 シリアル出力制御と送信リンク・パルス出力部

```
-- ***** シリアル出力 ***** --
TxDp     <= Tx_Data     when (Tx_Active = '1') else
            '1'         when (Tx_LinkPlus  = '1') else
            '0';

TxDn     <= not Tx_Data when (Tx_Active = '1') else
            '0'         when (Tx_LinkPlus  = '1') else
            '0';
    -- ↑送信アイドル時は両方とも"L"レベル

-- ***** 送信リンク・パルス出力 ***** --
process(CLK_20M, SYSRST)
    variable Link_Count : std_logic_vector(17 downto 0);
begin
    if (SYSRST = '1') then
        Tx_LinkPlus    <= '0';
        Link_Count  := (others => '0');
    elsif (CLK_20M'event and CLK_20M = '1') then
        if (Tx_Active = '1' Tx_Active = '1') then
            Tx_LinkPlus    <= '0';
            Link_Count  := (others => '0');
        else
            if (Link_Count = "111111111101110000") then
                                                    -- 約13.1ms経過
                Tx_LinkPlus    <= '1';
                Link_Count  := (others => '0');
            else
                if (Link_Count = "000000000000000001") then
                    Tx_LinkPlus    <= '0';
                end if;
                -- ↑リンク・パルス幅 2クロック(100ns)分
                Link_Count := Link_Count + '1';
            end if;
        end if;
    end if;
end process;
```

リスト 2　フレーム送信処理

```
-- ********** SEND_IDLE時の動作 ********** --
when SEND_IDLE => -- 送信アイドル時
    if (SEND_Start = '1' or Tx_ReStart = '1') then  -- 送信開始
        SEND_Active <= '1';            -- 送信開始(外部用)
        -- 送信イネーブル状態&送信最小ギャップ時間経過
        if (Tx_SendEnable = '1' and Tx_SendGap = '1') then
            Tx_Active <= '1';          -- 送信開始(内部用)
            Tx_Data <= '0';
            Bit_Count := "00000010";
            NSTATE := PREAMBLE;
        else                           -- 送信開始待ち
            NSTATE := SEND_WAIT;
        end if;
    else                               -- 送信アイドル
        NSTATE := SEND_IDLE;
    end if;
    iSEND_Collision    <= '0';
    Tx_SendRetry   <= '0';
    Tx_CRC32Clr        <= '0';

-- ********** SEND_WAIT時の動作 ********** --
when SEND_WAIT => -- 送信開始待ち
    -- 送信イネーブル状態&送信最小ギャップ時間経過
    if (Tx_SendEnable = '1' and Tx_SendGap = '1') then
        Tx_Active <= '1';        -- 送信開始(内部用)
        Tx_Data <= '0';
        Bit_Count := "00000010";
        NSTATE := PREAMBLE;
    else                         -- 送信開始待ち
        if (Tx_TimeOut = '1') then
            Tx_Active <= '0';  -- 送信終了
            SEND_SendErr <= '1';
            SEND_Active <= '0';
            NSTATE := SEND_IDLE;
        else
            NSTATE := SEND_WAIT;
        end if;
    end if;

-- ********** PREAMBLE時の動作 ********** --
when PREAMBLE =>  -- プリアンブル送出
    Tx_Data    <= Bit_Count(1);
    -- ビット・カウント処理
    if (Bit_Count = "01111100") then
        Bit_Count := (others => '0');
        NSTATE := FRAME_START;
    else
        Bit_Count := Bit_Count + '1';
        NSTATE := PREAMBLE;
    end if;
    SEND_SendErr   <= '0';

-- ********** FRAME_START時の動作 ********** --
when FRAME_START =>    -- フレーム・スタート送出
    Tx_Data    <= Bit_Count(0);
    -- ビット・カウント処理
    if (Bit_Count = "0000011") then
        Bit_Count := (others => '0');
        Byte_Count := SEND_Bytes;    -- 送信バイト数取得
                                 -- 送信データ(1バイト目)取り出し
        Tx_DATAtmp := SEND_Data;
                                 -- 次のアドレス計算
        iSEND_Address  <= iSEND_Address + '1';
        NSTATE := SEND_PACKET;
    else
        Bit_Count := Bit_Count + '1';
        NSTATE := FRAME_START;
    end if;

-- ********** SEND_PACKET時の動作 ********** --
when SEND_PACKET =>    -- パケット送信
    -- バイト→シリアル変換送信処理(LSBから送出)
    if (Tx_DATAtmp(0) = '1') then     -- 送信ビット'1'
        Tx_Data <= Bit_Count(0);      --"L"→"H"
    else                              -- 送信ビット'0'
        Tx_Data <= not Bit_Count(0); --"H"→"L"
    end if;
    -- 送信パケットCRC計算
    if (Bit_Count(0) = '0') then
        Tx_CRC32In   <= Tx_DATAtmp(0);
        Tx_CRC32Set <= '1';
    else
          Tx_CRC32Set <= '0';
    end if;
    -- 送信ビット/バイト・カウント
    if (Bit_Count = "00001111") then  -- 1バイト送信完了
        Bit_Count := (others => '0');
        if (Tx_SendEnable = '0') then -- 受信部動作中(衝突検出!)
            iSEND_Collision <= '1';
            NSTATE := SEND_FCS;
            -- ↑データ送信中の衝突検出はバイト単位境界で行う
        else
            if (Byte_Count = "00000000001") then -- 最後のバイト
                NSTATE := SEND_FCS;          -- 次からFCS送出
            else
                                -- 送信データ(次のデータ)取り出し
                Tx_DATAtmp := SEND_Data;
                                             -- 次のアドレス計算
                iSEND_Address <= iSEND_Address + '1';
                Byte_Count := Byte_Count - '1';
                NSTATE := SEND_PACKET;
            end if;
        end if;
    else -- 1バイト送信中
        if (Bit_Count(0) = '1') then           -- 奇数カウント時
                                               -- MSB→LSBシフト
            Tx_DATAtmp(6 downto 0) := Tx_DATAtmp(7 downto 1);
        end if;
        Bit_Count := Bit_Count + '1';
        NSTATE := SEND_PACKET;
    end if;

-- ********** SEND_FCS時の動作 ********** --
when SEND_FCS =>  -- FCS(CRC)送信
    if (iSEND_Collision = '0') then        -- 正常FCS送出
        -- FCS(CRC)送信処理(FCSはMSBから送出&ビット反転)
        if (Tx_CRC32Out(31) = '0') then  -- 送信ビット'0'
            Tx_Data <= Bit_Count(0);      --"L"→"H"の信号を出力
        else                              -- 送信ビット'1'
            Tx_Data <= not Bit_Count(0); --"H"→"L"の信号を出力
        end if;
    else                           -- ジャム信号送出(CRCエラーFCS)
        if (Tx_CRC32Out(31) = '1') then  -- 送信ビット'1'
            Tx_Data <= Bit_Count(0);      --"L"→"H"の信号を出力
        else                              -- 送信ビット'0'
            Tx_Data <= not Bit_Count(0); --"H"→"L"の信号を出力
        end if;
    end if;
    -- 送信パケットCRC計算結果シフト
    if (Bit_Count(0) = '0') then
        Tx_CRC32Shift <= '1';
    else
        Tx_CRC32Shift <= '0';
    end if;
    if (Tx_SendEnable = '0') then     -- 受信部動作中(衝突検出!)
        iSEND_Collision <= '1';
        -- ↑FCS送信中に衝突検出したら、残りのビットはジャム送信
    end if;
    -- 送信ビットカウント
    if (Bit_Count = "00111111") then -- 8バイト(32ビット)送信完了
        Bit_Count := (others => '0');
        NSTATE := SEND_END; -- FCS送出終了
    else -- 8バイト(32ビット)送信中
        Bit_Count := Bit_Count + '1';
        NSTATE := SEND_FCS;
    end if;

-- ********** SEND_END時の動作 ********** --
when others =>    -- パケット送信終了
    iSEND_Address <= (others => '0');
    Byte_Count         := (others => '0');
    if (Bit_Count = "00000101") then
        Tx_Data <= '0';
        Tx_Active <= '0';                  -- 送信終了
        -- パケット送信完了判定
        if (iSEND_Collision = '0') then  -- 衝突検出なし
            SEND_SendErr <= '0';
            SEND_Active <= '0';
            Tx_ColiCount <= (others => '0');
        else                                   -- 衝突検出あり
            if (Tx_ColiCount = "1111") then  -- 16回連続衝突
                SEND_SendErr <= '1';           -- 送信エラー
                SEND_Active <= '0';
                Tx_ColiCount <= (others => '0');
```

リスト 2　フレーム送信処理（つづき）

```
            else
                -- 再送信フラグ・セット
                Tx_SendRetry <= '1';
                Tx_ColiCount <= Tx_ColiCount
                                          + '1';
            end if;
        end if;
        Tx_CRC32Clr  <= '1';
        Tx_SendGapClr <= '0';
        Bit_Count := (others => '0');
        NSTATE := SEND_IDLE;
    else
        -- マンチェスタ符号の変化点をなくす
        Tx_Data <= '1';
        Tx_SendGapClr <= '1';
        Bit_Count := Bit_Count + '1';
        NSTATE := SEND_END;
    end if;
```

リスト 3　再送待ち処理

```
-- ***** パケット再送信待ち時間カウント処理 ***** --
process(CLK_20M, SYSRST, SEND_RESET)
    variable ReSendWait : std_logic;
    variable Count_A    : std_logic_vector(9 downto 0);
    variable Count_B    : std_logic_vector(9 downto 0);
begin
    if (SYSRST = '1' or SEND_RESET = '1') then
        Tx_ReStart <= '0';
        ReSendWait := '0';
        Count_A := (others => '0');
        Count_B := (others => '0');
    elsif (CLK_20M'event and CLK_20M = '1') then
        if (ReSendWait = '1') then        -- 再送待ちウェイト中
            if (Count_B = "0000000000") then  -- 待ち時間経過終了
                Tx_ReStart <= '1';        -- 再送開始
                ReSendWait := '0';
            else
                if (Count_A = "1111111111") then-- 51.2μs 経過
                    -- 単位時間カウンタ・デクリメント
                    Count_B := Count_B - '1';
                end if;
                Count_A := Count_A + '1';
            end if;
        else
            -- 衝突により送信が終了した
            if (Tx_Active = '0' and Tx_SendRetry = '1') then
                ReSendWait := '1';
                case Tx_ColiCount is
                    when "0001" =>
                        Count_B(9 downto 2) := (others => '0');
                        Count_B(1 downto 0) := Rx_RecvFCS(1 downto 0);
                    when "0010" =>
                        Count_B(9 downto 3) := (others => '0');
                        Count_B(2 downto 0) := Rx_RecvFCS(2 downto 0);
                                        ～中略～
                    when others =>
                        Count_B(9 downto 0) := Rx_RecvFCS(9 downto 0);
                end case;
                -- ↑受信パケット FCS 保持レジスタに残っている値を乱数がわりに使う
            end if;
            Tx_ReStart <= '0';
        end if;
    end if;
end process;
```

実現します．内部ロジックで‘1’を出力する場合はポジティブ側は“H”レベルでネガティブ側が“L”レベル，逆に‘0’を出力する場合はポジティブ側が“L”レベルでネガティブ側が“H”レベルを出力します．また，送信アイドル時は両方とも“L”レベルを出力します．

送信リンク・パルスは，13.1ms 周期で 100ns（20MHz クロックで 2クロック）のパルスを出力します．

● フレーム送信処理

リスト 2にフレーム送信処理部を示します．フレーム送信処理は全体をステート・マシンとして記述してみました．

送信開始信号がセットされたら，まず送信可能状態（受信部が受信動作をしていない）かを確認します．もし送信不許可状態であれば，SEND_WAIT ステートへ遷移し送信許可状態になるのを待ちます．送信許可状態であれば，すぐにプリアンブルの 1ビット目の前半の“L”レベルを出力します．

PREAMBLE ステートではプリアンブルを送信します．ここではビット・カウンタをカウント・アップし，ビット・カウンタのビット 1をそのままシリアル出力とします．SEND_IDLE ステートから遷移する段階でビット・カウンタに 2がセットされているので，“H”→“H”→“L”→“L”→…と 5MHz のクロックを送信することになります．これを SFD の直前まで送信します．

FRAME_START ステートでは SFD を送信します．ビット・カウンタのビット 0をそのまま出力することで，“L”→“H”→“L”→“H”を 4クロック分出力しています．これにより，マンチェスタ符号で‘1’を 2回連続で出力していることになります．

SEND_PACKET ステートでは，実際のデータ部分をシリアルに変換して出力します．また，CRC 演算回路にも入力して CRC の計算も開始します．データ部分は LSB から送信することになっているので，ビット 0から 1ビットずつ切り出しています．注意が必要なのは，このステートは 20MHz で動作し，マンチェスタ符号への変換もするので，次のビットにシフトする処理は 2回に 1回だけ，ビット・カウンタのビット 0が‘1’のときにだけ行っているという点です．

また，ビット・カウンタで 8ビット分処理したら，次の送信データを取り出して同様に送信します．同時にバイト・カウンタもカウント・ダウンします．バイト・カウンタが残り 1バイトになった時点で送信データはすべて送信したことになります．

そして，バイト単位境界で送信許許可状態を確認します．もし送信不許可状態ならパケットの衝突状態を意味しているので，衝突検出のフラグを立て，すべての送信データを送り出していなくても次の SEND_FCS ステートへ遷移します．

SEND_FCS ステートではデータ送信時に計算した CRC を，論理反転させて MSB から 1ビットずつ送信します．CRC 計算結果をシフトするのも，2回に 1回だけ行います．

なお，衝突検出のフラグがセットされていた場合は，CRC を論理反転させずに出力します．これにより，衝突時に送信したデータをだれかが受け取ってしまっていても，FCS が一致しないのでパケットを破棄させることができます．

SEND_END ステートはフレームの終了処理をします．マン

リスト 4　プリアンブル検出部

```
-- ***** プリアンブル部 5MHzクロック 検出 ***** --
process(CLK_100M, RECV_RESET, Rx_Active)
begin
    if (RECV_RESET = '1' and Rx_Active = '1') then
         Rx_PREAMBLE <= (others => '0');
    elsif (CLK_100M'event and CLK_100M = '1') then    -- 受信アイドル時のみサンプリング
         Rx_PREAMBLE(27 downto 1) <= Rx_PREAMBLE(26 downto 0);
         Rx_PREAMBLE(0) <= RxD;
    end if;
end process;

Rx_PreDetect <=           -- プリアンブル検出
    '1' when Rx_PREAMBLE(23 downto 21) = "000" and   Rx_PREAMBLE(20 downto 12) =   "111111111" and
                                                     Rx_PREAMBLE(11 downto 3) ="000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '1' when Rx_PREAMBLE(24 downto 22) = "000" and Rx_PREAMBLE(21 downto 13) =   "111111111" and
                                                     Rx_PREAMBLE(12 downto 3) ="0000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '1' when Rx_PREAMBLE(25 downto 23) = "000" and   Rx_PREAMBLE(22 downto 14) =   "111111111" and
                                                     Rx_PREAMBLE(13 downto 3) ="00000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '1' when Rx_PREAMBLE(24 downto 22) = "000" and   Rx_PREAMBLE(21 downto 12) =  "1111111111" and
                                                     Rx_PREAMBLE(11 downto 3) ="000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '1' when Rx_PREAMBLE(25 downto 23) = "000" and   Rx_PREAMBLE(22 downto 13) =  "1111111111" and
                                                     Rx_PREAMBLE(12 downto 3) ="0000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '1' when Rx_PREAMBLE(26 downto 24) = "000" and   Rx_PREAMBLE(23 downto 14) =  "1111111111" and
                                                     Rx_PREAMBLE(13 downto 3) ="00000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '1' when Rx_PREAMBLE(25 downto 23) = "000" and   Rx_PREAMBLE(22 downto 12) = "11111111111" and
                                                     Rx_PREAMBLE(11 downto 3) ="000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '1' when Rx_PREAMBLE(26 downto 24) = "000" and   Rx_PREAMBLE(23 downto 13) = "11111111111" and
                                                     Rx_PREAMBLE(12 downto 3) ="0000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '1' when Rx_PREAMBLE(27 downto 25) = "000" and   Rx_PREAMBLE(24 downto 14) = "11111111111" and
                                                     Rx_PREAMBLE(13 downto 3) ="00000000000" and Rx_PREAMBLE(2 downto 0) = "111" else
    '0';
```

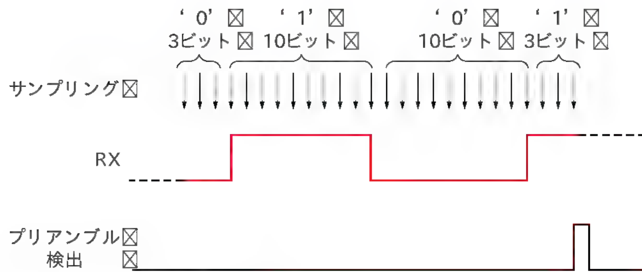


図8　プリアンブル検出の動作

チェスタ符号のビット中央のエッジの変化点をなくし，2ビット分以上の時間が経過するのを待ちます．

衝突検出時は衝突回数をカウントし，16回以内であれば，内部的に再送処理のフラグを立てます．再送を繰り返しても衝突が発生する場合には，送信エラーとして送信処理を完全に終了します．

● 再送待ち処理

リスト3（p.65）に再送待ち処理部を示します．送信終了時に再送フラグがセットされた場合は，51.2μs単位の時間待ちを何回か繰り返して再送までの時間を空けます．繰り返し回数は衝突回数によってビット幅を広げ，後述する受信FCS保持レジスタの値を取り込みます．

● 受信プリアンブル検出

次は受信部です．リスト4にプリアンブル検出部を示します．シリアル入力を28ビットのプリアンブル検出用シフト・レジスタに入力し，内部であらかじめもっているサンプリング・パターンに一致するかどうかをつねに比較しています．10Mbpsの信号を100MHzクロックでサンプリングしていくわけですが，クロックの誤差などを考慮して，“H”レベルおよび“L”レベルの期間を±1クロックずつのずれを考慮しています（図8）．

● マンチェスタ符号のエッジ検出＆データ復号

リスト5がマンチェスタ符号のエッジ検出＆データ復号部分です．エッジの検出を確実なものにするため，6ビットのエッジ検出用シフト・レジスタに入力していき，“L”レベルを3クロック，“H”レベルを3クロック検出したらエッジ検出信号をセットし，復号データを‘1’と判定します．逆の場合もエッジ検出信号をセットし，復号データを‘0’と判定します．判定タイミングは，前回のエッジを検出してから10±1クロック時間で行います（図9）．

マンチェスタ符号はビット中央で必ず信号変化をする符号なので，前回のエッジ検出から10クロック±1クロック期間のうちに，次のエッジがあるはずです．エッジを検出した場合にはカウンタをリセットし，次のエッジ検出まで待ちますが，カウンタが11クロックを超えた場合は，規定時間内にエッジが検出されなかったことを意味するので，マンチェスタ符号データの送信が終了した，つまりフレームが終わったと判断します．

● 受信データのシリアル→パラレル変換

リスト6（p.68）に受信データのシリアル→パラレル変換部を示します．受信アイドル状態からプリアンブルを検出するとフレーム受信開始状態になります．フレーム受信状態になったら，エッジ検出信号が立つたびにそのときのデータを調べ，‘1’が2回連続するSFDの検出を待ちます．SFDが検出されると，以降はデータ部分となります．ビット・カウンタをカウント・

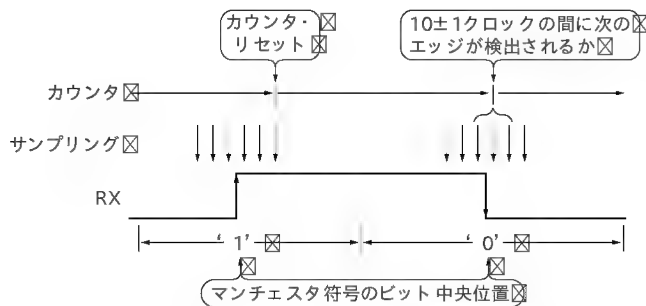


図9 マンチェスタ符号エッジ検出&データ復号部の動作

アップさせながら，復号されたシリアル受信データを受信データ用シフト・レジスタに入力していきます．データ部はLSBから送信されるので，受信側ではMSB側から入力&シフトしていきます．8ビット分カウントすれば1バイト受信完了で，受信バッファへデータをセットするとともに，受信バッファへの書き込み信号をセットします．またバイト・カウンタもカウント・アップします．受信バッファへの書き込みアドレスは，バイト・カウンタをそのまま使います．

受信データ・シリアル→パラレル変換は，フレーム受信状態が続くまで繰り返します．

● フレームのFCSチェック処理

送信時はシリアル出力時に1ビットずつ送信CRC演算回路に送信データをセットし，データ部の送信が終わったらその時点でのCRC演算回路の結果を反転させて32ビット分出力していきました．

受信時は，自分が受信したデータでCRCを演算するとともに，フレームの最後に送られてきたFCSを反転し，CRC演算結果と比較して値が一致するかどうかを確認することで，受信フレームの妥当性を確認します．つまり，受信時もCRC演算をする必要があるとともに，比較対象となるフレーム最後のFCSをうまく保持しなければならないわけです．

受信データのシリアル→パラレル変換部分では，受信フレームのFCSチェック用のために，FCS保持用の32ビット・シフト・レジスタへの入力と，受信フレームのCRC演算回路への復号されたシリアル受信データのセットも同時に行います．

ここで注意が必要なのは，フレームを受信している間は，FCSがどこから始まるのかがわからないという点です．マンチェスタ符号のエッジ検出がなくなったときがフレームの終了を意味しているので，フレームが終わったことを検出した時点から32ビット分だけ逆のぼらなければFCSの始まりがわからないのです．

そこで，フレームのはじめから1ビット受信するたびに，FCS保持用レジスタ全体をシフトしていき，フレームの終了時点で残った32ビットを受信したFCSとします．FCSはMSBから送信されるので，シフト・レジスタへはLSB側から入力&シフトしていきます．

また，CRC演算回路へデータをセットするタイミングも重要です．CRCの演算範囲はあくまでデータ部分のみです．しかし，すでに説明したように，フレーム受信中はFCSの開始位置がわかりません．つまり，1ビット・データ受信すると同時にCRC演算回路にも入力していたのでは，フレーム終了が判定できた

リスト5 マンチェスタ符号エッジ検出&データ復号部

```
-- ***** 受信状態時 RxD信号サンプリング ***** --
process(CLK_100M, RECV_RESET)
begin
    if (RECV_RESET = '1') then
         Rx_Shift <= (others => '0');
    elsif (CLK_100M'event and CLK_100M = '1') then
         Rx_Shift(5 downto 1) <= Rx_Shift(4 downto 0);
         Rx_Shift(0) <= RxD;
    end if;
end process;

-- ***** マンチェスタ符号エッジ&データ復号(ビットレート 10Mbps) ***** --
process(CLK_100M, RECV_RESET)
    variable Count : std_logic_vector(3 downto 0);
begin
    if (RECV_RESET = '1') then

         Rx_RecvEnd    <= '0';
         Rx_Active <= '0';
         Rx_Data       <= '0';
         Rx_Edg        <= '0';
         Count         := (others => '0');

    elsif (CLK_100M'event and CLK_100M = '1') then

         if (Rx_Active = '1') then              -- 受信開始
              -- 前回のエッジ検出から(9/10/11)-1カウント経過した
              if (Count = "1000" or Count = "1001"
                                 or Count = "1010") then
                   -- そろそろ次のエッジが検出されるはず…
                   if (Rx_Shift(5 downto 3) = "000" and
                       Rx_Shift(2 downto 0) = "111") then
                                                       -- ↑エッジ検出
                        Rx_Data  <= '1';  -- 受信データは'1'
                        Rx_Edg   <= '1';  -- エッジ検出ビット・セット
                        Count := (others => '0');
                   elsif (Rx_Shift(5 downto 3) = "111" and
                          Rx_Shift(2 downto 0) = "000") then
                                                       -- ↓エッジ検出
                        Rx_Data  <= '0';  -- 受信データは'0'
                        Rx_Edg   <= '1';  -- エッジ検出ビット・セット
                        Count := (others => '0');
                   else
                        Count := Count + '1';-- カウント・アップ動作
                   end if;
              else
                   Rx_Edg    <= '0';        -- エッジ検出ビット・クリア

                   -- エッジ検出がされないまま11-1クロックを超えた
                   if (Count > "1010") then
                        Count := (others => '0');
                        Rx_Active <= '0';  -- 受信動作終了
                        Rx_RecvEnd <= '1';
                   else
                        Count := Count + '1';-- カウント・アップ動作
                   end if;

              end if;

         else -- アイドル状態
              if (Rx_PreDetect = '1') then -- プリアンブル検出
                   Rx_Active <= '1';        -- 受信動作開始
              end if;
              Rx_RecvEnd <= '0';
         end if;

    end if;
end process;
```

リスト 6　受信データ・シリアル→パラレル変換

```
-- ***** 受信データ・シリアル→パラレル変換処理 ***** --
process(CLK_100M, RECV_RESET)
    variable Rx_Data_f : std_logic;
    variable DATA_Start : std_logic;
    variable CRCtmp     : std_logic_vector(31 downto 0);
begin
    if (RECV_RESET = '1') then

         iRECV_Data          <= (others => '0');
         iRECV_WR       <= '0';
         RECV_BitCount  <= (others => '0');
         RECV_ByteCount <= (others => '0');

         Rx_CRC32Clr         <= '1';
         Rx_CRC32In          <= '0';
         Rx_CRC32Set         <= '0';
         Rx_RecvFCS          <= (others => '0');

         Rx_Data_f      := '0';
         DATA_Start          := '0';

    elsif (CLK_100M'event and CLK_100M = '1') then

         if (Rx_Active = '1') then    -- 受信開始

              if (Rx_Edg = '1') then  -- エッジ検出
                   -- プリアンブル→SYNC待ち
                   if (DATA_Start = '0') then
                        RECV_ByteCount <= (others => '0');
                        RECV_BitCount  <= (others => '0');
                        Rx_CRC32Clr <= '1';
                        CRCtmp := "00000000000000000000000000000001";
                        -- '1'を2回連続で検出
                        if (Rx_Data_f = '1' and Rx_Data = '1') then
                             DATA_Start := '1';
                        end if;
                        Rx_Data_f := Rx_Data;-- 現在のデータを保存

                   else -- 送信データ部受信
                        -- MSB→LSBシフト
                        iRECV_Data(6 downto 0) <= iRECV_Data(7 downto 1);
                        iRECV_Data(7) <= Rx_Data;
                        -- 8ビット取り込み完了
                        if (RECV_BitCount = "111") then
                             RECV_BitCount <= (others => '0');
                             iRECV_WR <= '1';
                             RECV_ByteCount <= RECV_ByteCount + '1';
                        else
                             RECV_BitCount <= RECV_BitCount + '1';
                             iRECV_WR <= '0';
                        end if;

                        -- 受信パケットFCS保持
                        Rx_RecvFCS(0) <= Rx_Data;
                        Rx_RecvFCS(31 downto  1) <= Rx_RecvFCS(30 downto  0);
                        -- ↑パケット受信が終わった段階で残った32ビットが受信FCS

                        -- 内部CRC計算用
                        if (Rx_CRC32Clr = '0') then
                             -- FCS(CRC32)計算用ビット入力
                             Rx_CRC32In  <= CRCtmp(31);
                             -- FCS(CRC32)計算開始
                             Rx_CRC32Set <= '1';
                             -- ↑パケット受信開始33ビット目からCRC計算開始
                        end if;
                        if (CRCtmp(31) = '1') then
                             Rx_CRC32Clr <= '0';
                        end if;
                        -- ↑パケット受信開始から32ビット後にリセット解除
                        CRCtmp(31 downto 1) := CRCtmp(30 downto 0);
                        CRCtmp(0) := Rx_Data;  -- 32ビット遅延バッファ

                   end if;

              else
                   iRECV_WR <= '0';
                   Rx_CRC32Set <= '0';
              end if;

         else
              iRECV_WR <= '0';
              Rx_CRC32Set    <= '0';
              Rx_Data_f := '0';
              DATA_Start     := '0';
         end if;

    end if;
end process;
```

段階で，受信したFCSのビットまでCRC演算回路に入力してしまっていることになります．これでは正しいFCSにはなりません．

そこで，CRC演算回路に入力するデータを32ビット分遅延させ，フレームの終了が判定できた時点でデータ部だけのCRC演算結果が得られるようにします．また，CRC演算開始も32ビット分だけ遅らせる必要があるので，フレーム受信開始時に遅延バッファの最後のビットだけをセットしておき，遅延バッファから最初に'1'が出てくるのを待ち，その次のビットから演算を開始しています(図10)．

● 送信先MACアドレス取得

リスト7に受信フレームの送信先MACアドレス取得部を示します．フレームの受信開始でフラグをリセットしておき，1バイト受信するごとにMACアドレスを保持レジスタに格納していきます．1バイト受信が確定した時点でバイト・カウンタもカウント・アップされるので，MACアドレスの1バイト目はカウンタが"000"ではなく，"001"になる点に注意してください．

● 受信フレーム・エラー判定

リスト8に受信フレーム・エラー判定を行う部分を示します．

Ethernetでは，最小パケット・サイズ，および最大パケット・サイズの規定があります．FCSを含んだ場合は，最小が64バイト，最大が1518バイトとなります．フレーム終了時点でバイト・カウンタが最小サイズ未満，または最大サイズを超える場合は，フレーム・エラーとします．さらにビット・カウンタがゼロ以外の値だった場合は，半端なビット数を受信した状態でフレームが終わったことを意味しているので，これもフレーム・エラーとしています．

CRCエラーの判定は，内部で演算したCRC演算結果と，受信したFCSを反転した値を比較します．

これらはフレーム受信中も判定ロジックが常時動作しているので，フレームの受信中はほとんどの場合にはエラー状態を示すでしょう．よって，フレームの終了を判定した時点でこれらのエラー信号をサンプルし，必要であれば保持するようにします．

● 受信パケット・ステータス処理

リスト9に受信パケット・ステータス処理

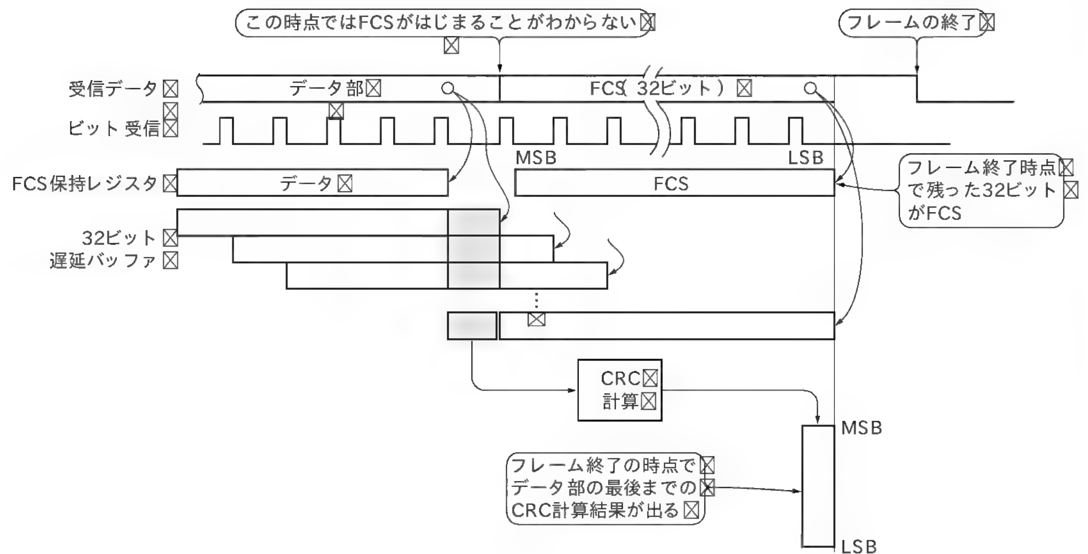


図10
受信フレームのFCSチェック方法

リスト7　送信先MACアドレス取得

```
-- ***** 受信パケット宛先MACアドレス取得 ***** --
process(CLK_100M, RECV_RESET)
    variable Flg   : std_logic;
begin
    if (RECV_RESET = '1') then
        RECV_PacketMAC <= (others => '0');
        Flg := '0';
    elsif (CLK_100M'event and CLK_100M = '1') then
        if (Rx_Active = '1') then     -- 受信開始
            -- 1バイト受信データ完了&送信先MACアドレス未取得時
            if (iRECV_WR = '1' and Flg = '0') then
                case RECV_ByteCount(2 downto 0) is
                    when "001" => -- 1バイト目
                        -- ↑1バイト受信した段階でバイト・カウンタが
                           進んでいるので
                        RECV_PacketMAC( 7 downto  0) <= iRECV_Data;
                    when "010" => -- 2バイト目
                        RECV_PacketMAC(15 downto  8) <= iRECV_Data;
                    when "011" => -- 3バイト目
                        RECV_PacketMAC(23 downto 16) <= iRECV_Data;
                    when "100" => -- 4バイト目
                        RECV_PacketMAC(31 downto 24) <= iRECV_Data;
                    when "101" => -- 5バイト目
                        RECV_PacketMAC(39 downto 32) <= iRECV_Data;
                    when "110" => -- 6バイト目
                        RECV_PacketMAC(47 downto 40) <= iRECV_Data;
                        Flg := '1';    -- 送信先MACアドレス取得終了
                    when others =>
                        null;
                end case;
            end if;
        else
            Flg := '0';
        end if;
    end if;
end process;
```

リスト8　受信フレーム・エラー判定

```
-- ***** 受信パケット CRC&フレーミング・エラー判定 ***** --
    RECV_FrameErr  <= iRECV_FrameErr;
    RECV_CRCError  <= iRECV_CRCError;
process(CLK_100M, RECV_RESET)
begin
    if (RECV_RESET = '1') then

        iRECV_FrameErr <= '0';
        iRECV_CRCError <= '0';

    elsif (CLK_100M'event and CLK_100M = '1') then

        if (
            (RECV_BitCount /= "000")
                -- 半端なビットがある!
        ) or (
            (RECV_ByteCount < "00001000000")
                -- 受信バイト数 64バイト未満
        ) or (
            (RECV_ByteCount > "10111101110")
                -- 受信バイト数 1518バイトを超える
        ) then
            iRECV_FrameErr <= '1';
                        -- フレーム・エラー検出
            iRECV_CRCError <= '0';
            -- ↑フレームエラー検出を優先する

        elsif (Rx_RecvFCS /= not Rx_CRC32Out) then
                        -- 一致しない!!
            iRECV_FrameErr <= '0';
            iRECV_CRCError <= '1';
                        -- CRCエラー検出

        else
            iRECV_FrameErr <= '0';
            iRECV_CRCError <= '0';
        end if;

    end if;
end process;
```

部を示します．フレームが終了した時点で，各種受信モードにしたがって受信完了信号や各種エラー信号をセットする部分です．

受信モードとして，自分以外の宛て先のパケットもすべて受信するという場合は，宛て先MACアドレスの比較は不要なので，フレームの受信終了が即座にパケット受信完了を意味します．通常は，送信先MACアドレスと自分のMACアドレスが一致した場合と，送信先アドレスがブロードキャスト・アドレスだった場合に，パケット受信完了信号をセットします．

さらに，エラーがあってもとりあえず受信できたパケットは受信バッファに格納するという動作モードを用意したので，それらのモードが設定されていた場合には，CRCエラーやフレーム・エラーがあっても受信完了信号をセットします．

リスト 9
受信パケット・ステータス処理

```
-- ***** 受信パケット・ステータス判定処理 ***** --
process(CLK_100M, RECV_RESET)
begin
    if (RECV_RESET = '1') then

         RECV_DataSetRdy <= '0';
         RECV_RecvPacket <= '0';

    elsif (CLK_100M'event and CLK_100M = '1') then

         if (Rx_RecvEnd = '1') then   -- パケット受信完了

              -- 送信先MACアドレスを識別しない場合
              if (RECV_AllPacket = '1') then

                   -- ↓受信パケットの送信先アドレスは比較する必要なし
                   -- 自分が処理すべきパケットを受信した場合
                   if (
                        (iRECV_FrameErr = '0' and iRECV_CRCError = '0')
                   ) or (
                        -- CRCエラー・パケットも受信バッファに格納
                        (RECV_CRCErrBuf = '1' and iRECV_CRCError = '1')
                   ) or (
                        -- フレーム・エラー・パケットも受信バッファに格納
                        (RECV_FrmErrBuf = '1' and iRECV_FrameErr = '1')
                   ) then
                        -- 受信バッファに有効なデータを格納した
                        RECV_DataSetRdy <= '1';
                   else
                        RECV_DataSetRdy <= '0';
                        -- ↑自分が処理すべきパケットだったが、
                        -- エラーにより受信バッファを破棄する
                   end if;
                   RECV_RecvPacket <= '1'; -- パケットを受信した

              else -- 送信先MACアドレスを識別する場合
                   if (
                        (RECV_ByteCount > "00000000101")
                        -- ↑6バイト以上受信しないと宛先MACアドレスが不明
                   ) and (
                        (
                             -- 自分のMACアドレスと一致
                             (RECV_PacketMAC = MAC_Address)
                        ) or (
                             -- ブロードキャスト・パケットも受け取る&
                             (RECV_NotBroadC = '0' and
                             -- ブロードキャスト・アドレスが一致
                                           RECV_PacketMAC = X"FFFFFFFFFFFF")
                        )
                   ) then    -- 自分が処理すべきパケットを受信した場合
                        if (
                             (iRECV_FrameErr = '0' and iRECV_CRCError = '0')
                        ) or (
                             -- CRCエラー・パケットも受信バッファに格納
                             (RECV_CRCErrBuf = '1' and iRECV_CRCError = '1')
                        ) or (
                             -- フレーム・エラー・パケットも受信バッファに格納
                             (RECV_FrmErrBuf = '1' and iRECV_FrameErr = '1')
                        ) then
                             -- 受信バッファに有効なデータを格納した
                             RECV_DataSetRdy <= '1';
                        else
                             RECV_DataSetRdy <= '0';
                             -- ↑自分が処理すべきパケットだったが、
                             -- エラーにより受信バッファを破棄する
                        end if;
                        RECV_RecvPacket <= '1'; -- パケットを受信した

                   end if;

              end if;
         else
              RECV_DataSetRdy <= '0';
              RECV_RecvPacket <= '0';
         end if;

    end if;
end process;
```

リスト10　CRC演算部

```
-- ***** CRC計算結果出力 ***** --
    CRC32Out <= Data;

-- ***** CRC計算 ***** --
process(CLK, CRC32Clr)
    variable fb : std_logic;
begin
    if (CRC32Clr = '1') then
        Data <= (others => '1'); -- 初期値オール'1'
        fb      := '0';
    elsif (CLK'event and CLK = '1') then
        if (CRC32Shift = '1') then
            -- ビット・シフト出力
            Data(32 downto 2) <= Data(31 downto 1);
            Data(1) <= '1';
        elsif (CRC32Set = '1') then
            -- CRC演算
            fb := CRC32In xor Data(32);  -- X^32
            Data(32 downto 28) <= Data(31 downto 27);
            Data(27) <= Data(26) xor fb; -- X^26
            Data(26 downto 25) <= Data(25 downto 24);
            Data(24) <= Data(23) xor fb; -- X^23
            Data(23) <= Data(22) xor fb; -- X^22
            Data(22 downto 18) <= Data(21 downto 17);
            Data(17) <= Data(16) xor fb; -- X^16
            Data(16 downto 14) <= Data(15 downto 13);
            Data(13) <= Data(12) xor fb; -- X^12
            Data(12) <= Data(11) xor fb; -- X^11
            Data(11) <= Data(10) xor fb; -- X^10
            Data(10) <= Data( 9);
            Data( 9) <= Data( 8) xor fb; -- X^8
            Data( 8) <= Data( 7) xor fb; -- X^7
            Data( 7) <= Data( 6);
            Data( 6) <= Data( 5) xor fb; -- X^5
            Data( 5) <= Data( 4) xor fb; -- X^4
            Data( 4) <= Data( 3);
            Data( 3) <= Data( 2) xor fb; -- X^2
            Data( 2) <= Data( 1) xor fb; -- X^1
            Data( 1) <= fb;              -- X^0=1
        end if;
    end if;
end process;
```

● CRC演算部

リスト10にCRC演算部分を示します．フレームの送信開始およびフレームの受信開始時にリセット信号が入力されるので，CRC演算開始時点では出力レジスタはすべて'1'を示します．

CRC演算は1ビットずつ行います．CRC演算データがセットされると，仕様で規定されたCRC32の計算式に従い，特定のビットを排他的論理和をしながら全体をMSB側にシフトします．LSBには入力データとMSBのビットを排他的論理和をした値を入れます．

フレーム受信時は最後に32ビットを一度に比較するので，出力レジスタは32ビット分を上位モジュールにも出力します．フレーム出力時は1ビットずつ出力していくので，結果をMSB側にシフトしていく処理もこのモジュール内に実装しました．

まとめ

以上，Ethrenetコントロール部分について詳しく解説してきました．本来なら，システム・バスへの接続部分についても詳しく解説し，各種制御信号をどのようにシステム・バスにマッピングするか，受信完了信号をセットしたときに割り込みを発生させる処理などについても解説したかったのですが，誌面の関係からここで止めておきます．

今回作成したEthrenetコントローラは，PCIバス制御部分も含めてHDLソースを公開します．次に本誌にCD-ROMが付属する号(InterGiga No.33)で，プロジェクト・ファイルを含む全ファイルを収録する予定です．それまでに，設計をもう少しブラッシュ・アップしたいとも考えています．ご期待ください．

まつもと・のぶゆき　(株)タムラ製作所
やまたけ・いちろう　来栖川電工(有)

column 3
受信フレームのFCSチェック方法

今回は受信フレームのFCSチェック方法として，自前で計算しつつ最後に比較するという方法を採っています．

このようなデータの妥当性を確認する方法として，たとえばチェック・サムのような場合は，オーバフローは無視してすべての値を加算し，残った結果がゼロである場合に正しいと判定します．これと同様にEthrenetでも，FCS部分までいっしょに計算処理をすることで，フレーム終了の時点でオール・ゼロなりオール'1'なりの結果が得られるような方法はないものか検討したのですが，良い方法を思いつきませんでした．

そのため，内部でのCRC演算用には32ビットの遅延バッファや，演算開始を32ビット分遅らせる処理などが必要になってしまいました．「こうやるともっと簡単だよ！」というアイデアをお持ちの方は，ぜひ編集部までお知らせいただき，解説記事を執筆していただけるとありがたいと思います．

第**4**章



ほかのLANカードやハブと通信できるかパケットの品質を確認する

10Base-T対応LANカードの動作確認試験

松本 信幸/山武 一朗

ここでは第3章で設計したオリジナルの10Base-T対応LANカードの動作確認の方法について解説する．実際に機器と接続するためには，送受信信号が電気的に問題ないかどうか，そして実際にパケットを流してエラーなくデータのやりとりができるかどうかを確認する必要がある．また，通信テストであるからには，接続する相手も必要である．パケット送受信テストで必要な機材やテスト・プログラムなどについても解説する．　（編集部）

1 評価環境について

● オリジナル10Base-T対応LANカードの評価内容

今回作成したカードの評価は，大きく二つの観点から行います．一つ目は電気的なもので，オシロスコープやロジック・アナライザを用いて波形を観測するというものです．今回のカードはFPGAを用いて行うという学習目的のものであるため，IEEE802.3の仕様を完全に満たすことを確認するわけではなく，実際どこまで使用できるかを見てみる程度になります．

そして二つ目の評価こそ本命で，端的にいってほかのLANカードやハブと正しく通信できるかどうかを見るというものです．

● パケット送受信テスト方法

通信のテストをするからには，自分以外に相手も必要になります．相手側には，こちらが送ったパケットを相手が受け取り，それを表示する機能，そして相手から任意のパケットをこちらに送る機能です．

筆者は普段はSnifferを使っていますが，非常に高価な機材なので，読者のだれもが気軽に使えるものではありません．そこで今回は，Ethernet用アナライザ・ソフトウェア PacMon（http://www.layer.co.jp/）を使ってみました．PacMonのプロ版にはパケット送信機能もあるので，任意のパケットの送信テストに使えます．流れてくるパケットを見てみるだけなら，スタンダード版でもいいでしょう．

しかし，PacMonのようなソフトウェア的なLANアナライザではテストできない内容もあります．たとえば，こちらから送ったパケットにフレーム・エラーやCRCエラーのようなエラーがあった場合，たいていのLANカードではそのパケットを捨ててしまうので，ソフトウェア的なLANアナライザでは見ることができないのです．また，テストのためにあえてエラーのあるパケットを送信するということもできません．

そこで今回は，NE2000互換LANカードの制御レジスタを直接制御して，エラーのあるパケットも受信したり(受信できない場合もあるが)，また最小パケット・サイズ以下のパケットを送ってフレーム・エラーが出るかどうかなど(NE2000では60バイト未満のバイト数のパケットでもそのままのバイト数で送信可能)の確認ができるような，パケット送受信テスト・プログラムも用意してテストしてみます．

また，今回製作したLANカードからパケットを送ったり，受け取ったパケットのステータスやデータをダンプ表示する必要もあるので，そのためのテスト・プログラムも作成しました．

2 電気的動作の確認

電気的な動作の確認は，送信側と受信側の両方で，それぞれ行うことにします．また，10Base-Tの場合，パケットの送受信だけでなく，アイドル時にはリンク・パルスの送受信も行われるので，これらの波形についても確認する必要があります．

また，10Base-Tの最大ケーブル長は100mです．そこで数mの長さのケーブルと，50mのケーブル2本を間に中継器を入れてつないで100mとしたものを用意して，終端の波形の違いを見てみることにします．

● 送信アイドルと送信リンク・パルスの確認

10Base-Tでは，送信アイドル状態の差動出力は±50mV以下という規定があります．

また，IEEE802.3の規格書における第13章と第14章の10Base-Tの項目では，リンク・パルスなどの規定はありますが(電圧500mV～3.1V)，送出の電位などについて明確な規定は書かれていません．さらにこのリンク・パルスについても，"H"レベルの幅が50ns程度のパルスを16±8ms間隔で送出するという，時間軸の桁が違うものを用いています．パルス間隔を最短の8msとしても，50nsパルスから見て16万倍の大きさとなります．

このような波形でも，アナログのオシロスコープでトリガを

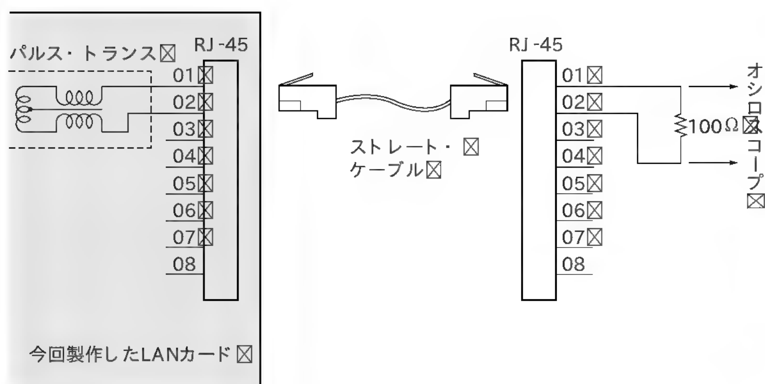


図1 送信リンク・パルスの測定方法



写真1 送信リンク・パルスの終端電圧波形(上：100mケーブル，下：5mケーブル)

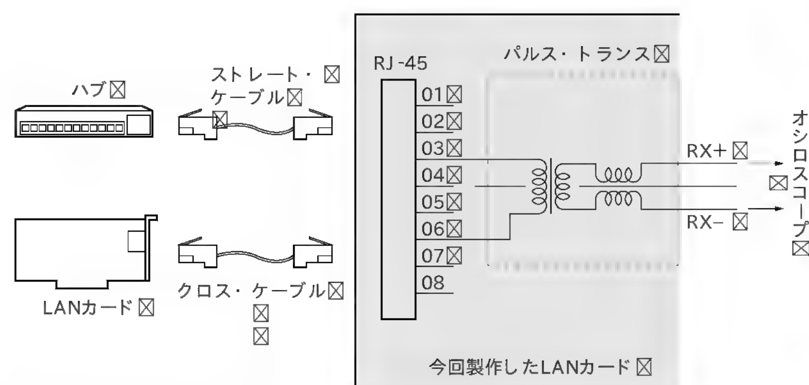


図2 受信リンク・パルスの測定方法

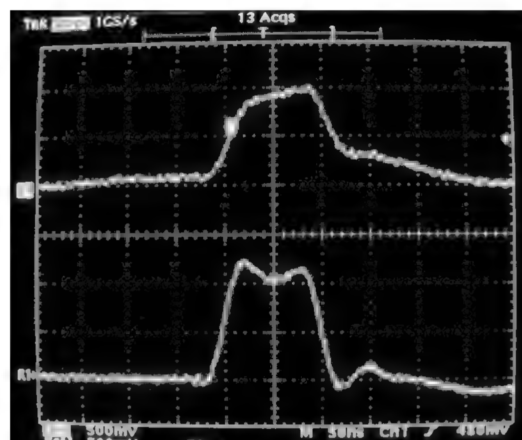


写真2 受信リンク・パルスの波形(上：100mケーブル，下：5mケーブル)

うまく引っ掛ければ波形を見ることもできますが，少しテクニックがいるで，ここではディジタル・ストレージ・オシロスコープを用意して波形を観測してみます．

まず図1の測定回路で，送信リンク・パルスの信号を見たものが写真1です．上が100mケーブル時，下が5mケーブル時の終端電圧波形です．リンク・パルス幅はここでは100nsとしています．5mケーブル時は1V以上の電圧が出ていますが，100mともなるとかなり波形がなまり，電圧もぎりぎり1Vといったところです．

送信アイドル時についても，5mケーブル時のオーバ・シュート/アンダ・シュート部分が多少気になりますが，このくらいならまったく問題ないと思われます．

また，どうしても周波数帯域の低いアナログ・オシロスコープしか用意できない場合は，たとえばFPGAの中身の設計を変更して，送信リンク・パルスの送出間隔を数百nsオーダにまで短くして波形を見る方法もあるでしょう．

それでは，このケーブルを10Base-Tのハブにつないで見ましょう．無事，ハブのリンクLEDが点灯したでしょうか．また同時に，設計したLANカード側でもリンクのLEDが点灯しているかと思います．

実は当初はリンク・パルスの幅を50nsにしていたのですが，手元のRTL8019ASを搭載したLANカードとクロス・ケーブルで直結した場合に，リンクLEDがときどき点滅してリンクが切れる場合が見られました．いろいろテストしたところ，リンク・パルスの幅を100nsにしたら安定して動作するようになったので，今回はパルス幅を100nsとしています．

● 受信リンク・パルスの確認

次に受信リンク・パルスの状態を確認してみましょう．すでにハブに接続した段階で，自分のリンクLEDも点灯していると思いますが，このとき，具体的にどんな波形が入力されているかを見てみます．

図2が受信リンク・パルスの測定方法です．パルス・トランスのFPGA側のRX＋とRX－の間にプローブを当ててみた場合の波形が写真2です．これも，上が100mケーブル経由時の波形で，下が5mケーブル時の波形です．

● リンクLEDはあまりあてにならない!?

パケットの送受信を行うには，まず最初にリンクが立たなけ

ればなりません．今回のインターフェース・カードをハブに接続するならば，ハブに用意されているリンクLEDが点灯するかどうかが最初の確認事項になります．しかし，リンクLEDを点灯させるために用いるリンク・パルスについては，かなり大ざっぱな信号でもよく，何らかの信号が定期的にさえ（最大24ms間隔で）到着すればリンクされているものとみなされてしまいます．このため，ハブのリンクLEDが点灯したからといって，安心はできません．「点灯して当たり前」くらいに考えておかなければなりません．

● **送信パケット波形の確認**

次は実際にパケットを送受信してみます．リンク・パルスは何もしなくてもやりとりされるものですが，パケットの送受信はそれを制御するプログラムが必要になります．

リスト1（pp.75-76）に，今回製作したLANカード用のパケット送信テスト・プログラムを示します．とりあえず，パケットの送信テストなので，MACアドレスの設定や割り込みまわりの初期化などは不要です．デバイス初期化関数を呼んで，送信イネーブル状態にした後は，送信バッファが空いたら送信開始ビットをセットして，連続してパケット送信を繰り返すだけです．

図3に送信パケット波形の測定方法を示します．相手のLANカードのパルス・トランスの受信側にプローブを当てます．まずは送信差動信号の波形の対称性を見てみましょう．パルス・トランスのセンタ・タップをGNDクリップでつまみ，RX＋とRX－にそれぞれプローブを当てて測定したのが写真3です．こ

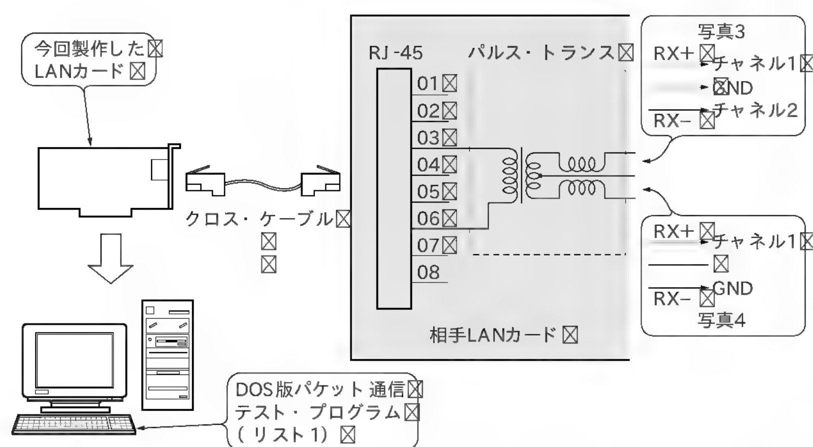


図3　送信パケット波形の測定方法

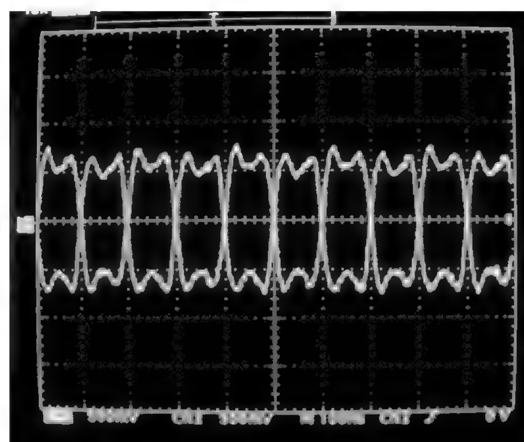


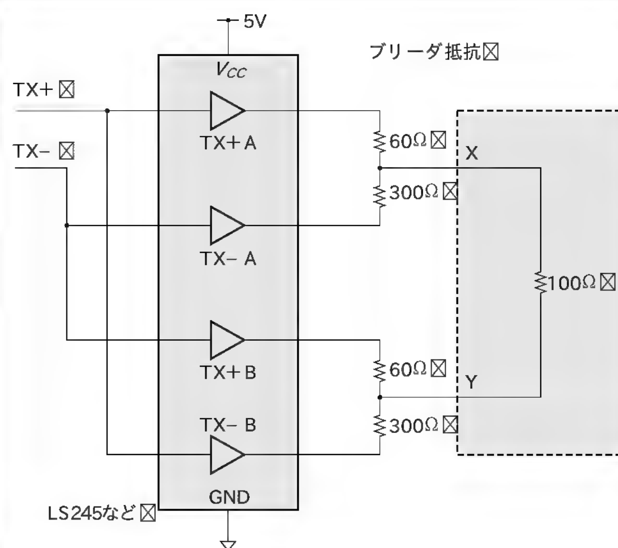
写真3　送信差動信号の対称性の確認

column 1

送信部に5V系TTLを使う場合

今回使用したFPGAはI/O電圧が3.3V系なので，ダンピング抵抗を入れた程度でほとんどパルス・トランス直結状態で使えますが，5V系のI/O電圧をもつCPLDやTTL，もしくはCMOSデバイスを使う場合には，図Aのようなブリーダ抵抗を使用して，見かけ上3V程度の差動信号に似た出力を作ることにより，送信部分を構成すること可能です．

これは，差動信号であるTX＋，TX－に関して，TX＋Aに対してTX＋を入力すると同時にTX－Bにも接続し，同様にTX－AにTX－を接続すると同時にTX＋Bにも接続します．TX＋Aが“H”レベルの場合，出力電位はおよそ4.8Vあります．これに対してTX＋Bは“L”レベルなので0Vとなります．この出力間にブリーダ抵抗が60Ωと300Ωで接続されているので，X点の電位は4.0Vとなります．TX－はTX＋の逆になるので，TX－Aは“L”レベル，TX－Bは“H”レベルとなり，Y点の電位は0.8Vとなります．このことからX-Y間の電位差は3.2Vとなるので，LV-TTLと同程度の電位で出力を行うことが可能です．



図A　5V系TTLを使う場合の回路

リスト1 送信テスト・プログラム(一部)

```
/*****************************************************************
オリジナル仕様ネットワーク・コントローラ IF NIC ドライバ部
*****************************************************************/

～中略～
/* シリアルROM読み出し */
void serial_rom_read( int adr, int len, UBYTE *buff)
{
    int i;
    ULONG l;
    if ( len>256) len=256;  /* 最大256バイト */
    for( i=0;i<len;i++) {
        l=i;
        out_long( ROM_Ctrl,ROM_AccStart|ROM_Read|(( l+adr)
<<16)) ;
        for( ;;) {      /* Busyビット・クリア待ち */
            l=in_long( ROM_Ctrl) ;
            if (( l&ROM_AccBusy) ==0) break;
        }
        l=in_long( ROM_Ctrl) ;
        *buff=( UBYTE) l;    /* 下位8ビット読み出しデータ */
        buff++;
    }
}

/* MACアドレス設定( NICレジスタへの設定) */
void NIC_SetMACaddress( UBYTE *p)
{
    int i;
    UBYTE c;
    ULONG l;
    for( i=0;i<6;i++) {
        l=i;
        c=*p;
        out_long( MAC_Ctrl,MAC_SetEnable|( l<<16) |c) ;
        p++;
    }
}

/* MACアドレス取得( NICレジスタからの取得) */
void NIC_GetMACaddress( UBYTE *p)
{
    int i;
    ULONG l;
    for( i=0;i<6;i++) {
        l=i;
        out_long( MAC_Ctrl,l<<16) ;
        l=in_long( MAC_Ctrl) ;
        *p=( UBYTE) l;
        p++;
    }
}

/* NIC初期化 */
void NIC_Init( void)
{
    UBYTE buff[6];

    /* 送受信部リセット */
    out_long( RECVx_Conf,RECVx_Reset) ;
    out_long( SENDx_Conf,RECVx_Reset) ;
    out_long( SEND0_Ctrl, 0) ;

    /* シリアルROM( アドレス10h) からMACアドレス読み出し */
    serial_rom_read( 0x10, 6, buff) ;
    /* MACアドレス設定( NICレジスタへの設定) */
    NIC_SetMACaddress( buff) ;

    /* 送受信部リセットクリア */
    out_long( RECVx_Conf,0) ;
    out_long( SENDx_Conf,0) ;

    /* 各種変数初期化 */
    SendBuff_Send=0;
    Packet_RecvReady=0;
    RecvError=0;
    SendError=0;
    Packet_BuffSel=0;
}


/* NIC割り込み初期化 */
void NIC_Interrupt_Init( void)
{
    /* 送信系エラー割り込み/送信完了割り込み/受信系エラー割り込み/受信バッファ0～
3完了割り込み 割り込みクリア */
    out_long
( IFnic_ISR,SENDx_ErrSta|SEND0_IntSta|RECVx_ErrSta|RECV3_IntSta|RE
CV2_IntSta|RECV1_IntSta|RECV0_IntSta) ;
    /* 送信系エラー割り込み/送信完了割り込み/受信系エラー割り込み/受信バッファ0～
3完了割り込み 割り込みマスク解除 */
    out_long
( IFnic_IMR,SENDx_ErrMsk|SEND0_IntMsk|RECVx_ErrMsk|RECV3_IntMsk|RE
CV2_IntMsk|RECV1_IntMsk|RECV0_IntMsk) ;
}

/* NIC割り込み処理 */
void NIC_Interrupt( void)
{
    ULONG Status;
    Status=in_long( IFnic_ISR) ;  /* 割り込みステータス取得 */
    out_long( IFnic_ISR,Status) ; /* 割り込みステータス・クリア */

    if ( Status&RECV0_IntSta)  {  /* 受信バッファ0受信完了割り込み */
        Packet_RecvReady=Packet_RecvReady|RECV0_IntSta;
    }
    if ( Status&RECV1_IntSta)  {  /* 受信バッファ1受信完了割り込み */
        Packet_RecvReady=Packet_RecvReady|RECV1_IntSta;
    }
    if ( Status&RECV2_IntSta)  {  /* 受信バッファ2受信完了割り込み */
        Packet_RecvReady=Packet_RecvReady|RECV2_IntSta;
    }
    if ( Status&RECV3_IntSta)  {  /* 受信バッファ3受信完了割り込み */
        Packet_RecvReady=Packet_RecvReady|RECV3_IntSta;
    }
    if ( Status&RECVx_ErrSta)  {  /* 受信系エラー割り込み */
        RecvError=RECVx_ErrSta;
    }

    if ( Status&SEND0_IntSta)  {  /* 送信完了割り込み */
        SendBuff_Send=0;          /* 送信完了 */
    }
    if ( Status&SENDx_ErrSta)  {  /* 送信系エラー割り込み */
        SendError=SENDx_ErrSta;
    }
}

/* 送信バッファ空き状態チェック */
int NIC_SendBufferCheck( void)
{
    if ( SendBuff_Send==0)  {
        return 0;      /* 送信バッファが空いている */
    } else {
        return -1;     /* 送信バッファが空いていない */
    }
}

/* パケット送信 */
int NIC_SendPacket( UBYTE *buff, UWORD len)
{
    int i,end;
    ULONG *rbuf,wbuf,l;
    if ( SendBuff_Send==0)  {      /* 送信バッファが空いている */
        if ( len<64)    len=64;    /* 最小64バイト */
        if ( len>1514)  len=1514;  /* 最大1514バイト */
        end=len>>2;                /* 32ビットPIO転送なので1/4にする */
        if ( len&3)  end++;        /* 端数があればもう1ワード */
        rbuf=( ULONG *) buff;
        wbuf=SEND0_Buff;
        for( i=0;i<end;i++) {
            l=*rbuf;
            out_long( wbuf,l) ;    /* 送信バッファ書き込み */
            wbuf=wbuf+4;           /* ロング・サイズ */
            rbuf++;
        }
        out_long( SEND0_Ctrl,SENDx_Start|len) ;
                                   /* 送信開始/送信バイト数 */
        SendBuff_Send=1;           /* 送信バッファ送信処理 */
        return 0;                  /* 送信バッファ格納完了 */
    } else {
        return -1;                 /* 送信バッファが空いていない */
    }
}
```

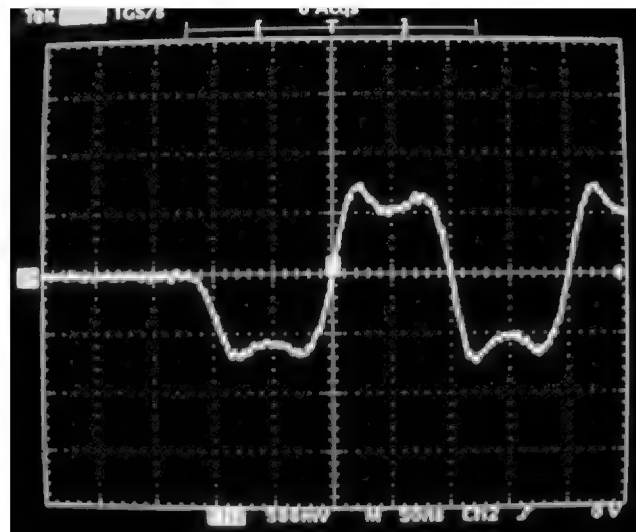
リスト 1　送信テスト・プログラム(一部)

```
/* 送信バッファ再送信 */
void NIC_SendRetry( void)
{
    ULONG len;
    len=in_long( SEND0_Ctrl) &SENDxBytes;  /* 送信バイト数取得 */
    out_long( SEND0_Ctrl,SENDx_Start|len) ;/* 送信開始/送信バイト数 */
}

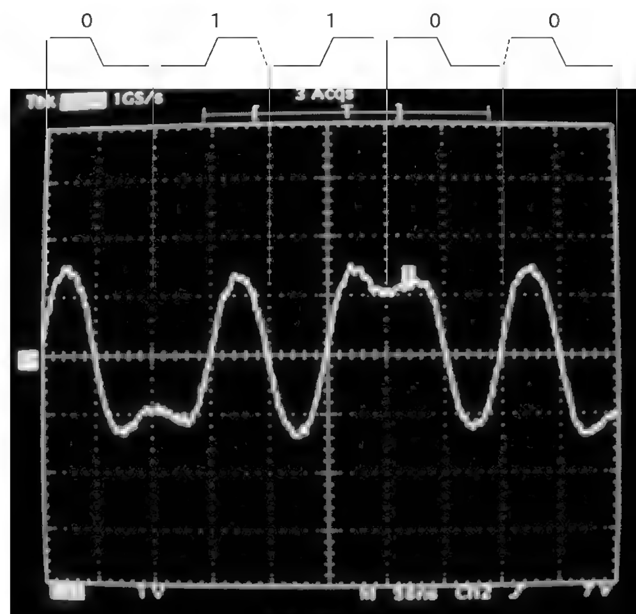
/* パケット受信 */
UWORD NIC_RecvPacket( UBYTE *buff)
{
    int i,end;
    UWORD len;
    ULONG rbuf,*wbuf,ctrl,l;

    switch( Packet_BuffSel) {
        case 0:
            ctrl=RECV0_Ctrl;
            rbuf=RECV0_Buff;
            break;
        case 1:
            ctrl=RECV1_Ctrl;
            rbuf=RECV1_Buff;
            break;
        case 2:
            ctrl=RECV2_Ctrl;
            rbuf=RECV2_Buff;
            break;
        default:
            ctrl=RECV3_Ctrl;
            rbuf=RECV3_Buff;
            break;
    }
    l=in_long( ctrl) ;         /* 受信バッファ x ステータス取得 */
    if ( l&RECVx_Req)  {       /* 受信完了 */
        wbuf=( ULONG *) buff;
        len=( UWORD)( l&RECVx_Bytes) ; /* 受信バイト数取り出し */
        end=len>>2;            /* 32 ビット PIO 転送なので 1/4 にする */
        if ( len&3)  end++;  /* 端数があればもう 1 ワード加算 */
        for( i=0;i<end;i++) {
            *wbuf=in_long( rbuf) ;    /* 受信バッファ読み出し */
            wbuf++;
            rbuf=rbuf+4;              /* ロング・サイズ */
        }
        switch( Packet_BuffSel) {
            case 0:
                Packet_BuffSel=1;   /* 次の受信バッファへ */
                Packet_RecvReady=Packet_RecvReady&0xfffe;
                break;
            case 1:
                Packet_BuffSel=2;   /* 次の受信バッファへ */
                Packet_RecvReady=Packet_RecvReady&0xfffd;
                break;
            case 2:
                Packet_BuffSel=3;   /* 次の受信バッファへ */
                Packet_RecvReady=Packet_RecvReady&0xfffb;
                break;
            default:
                Packet_BuffSel=0;   /* 次の受信バッファへ */
                Packet_RecvReady=Packet_RecvReady&0xfff7;
                break;
        }
        out_long( ctrl,RECVx_Ack) ;   /* 受信データ取り出し完了 */

    } else {
        /* 受信割り込みが発生したのに受信完了フラグが立っていない */
    }
    return len;
}
```



(a) プリアンブル部



(b) データ部

写真 4　送信パケット波形

れはプリアンブル部分を見ています．周波数が 5MHz の，差動動作による波形がきれいに出ているのがわかります．この対称性のバランスが崩れていると，うまくパケットを受け取ってもらえません．

次にパルス・トランスの RX－を GND クリップでつまみ，プローブを RX＋にあててプリアンブルの先頭とデータ部分の波形を見てみましょう．

写真 4(a)がプリアンブルの先頭部分のようす，写真 4(b)がデータ部分の波形です．プリアンブルでは 100ns ごとに波形の山と谷が入れ替わっていて，ちょうど周期 200ns(5MHz)の方形波が出ているようすがわかります．データ部では，いちばん左側の波形の立ち下がり位置をマンチェスタ符号の'0'のビットの中央だとすると，100ns 後の立ち上がりは'1'を，次の 100ns 後の立ち上がりも'1'，さらに次の立ち下がりは'0'を示していることがわかります．

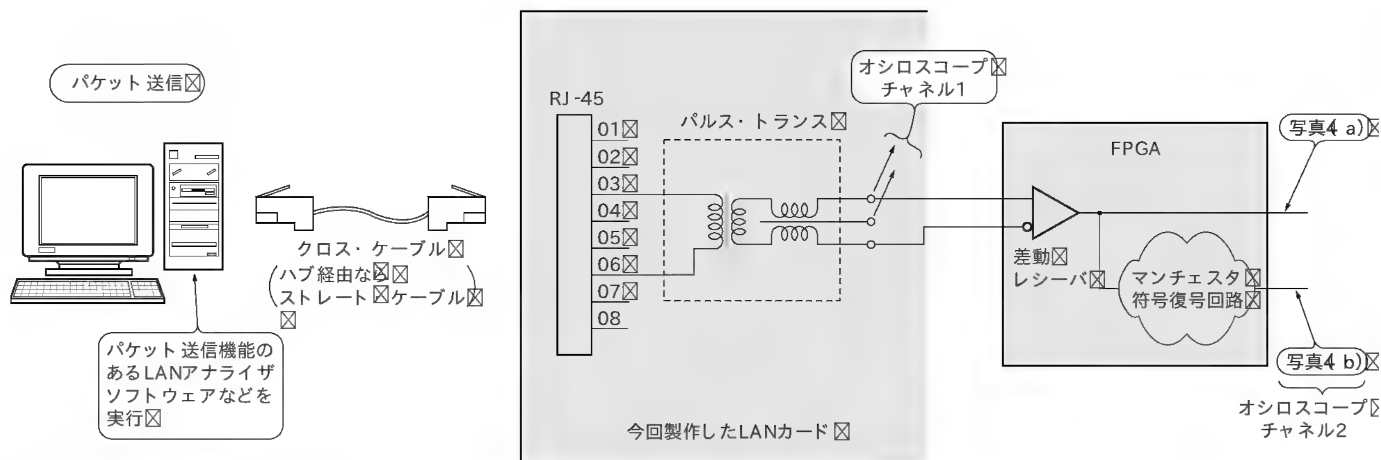


図4　受信パケット波形の測定方法

● 受信パケット波形の確認

次は受信側です．パケットの受信テストでは，今回はLANアナライザ・ソフトウェアを使い，任意のパケット・データを数ms間隔で繰り返し送信してみることにします．

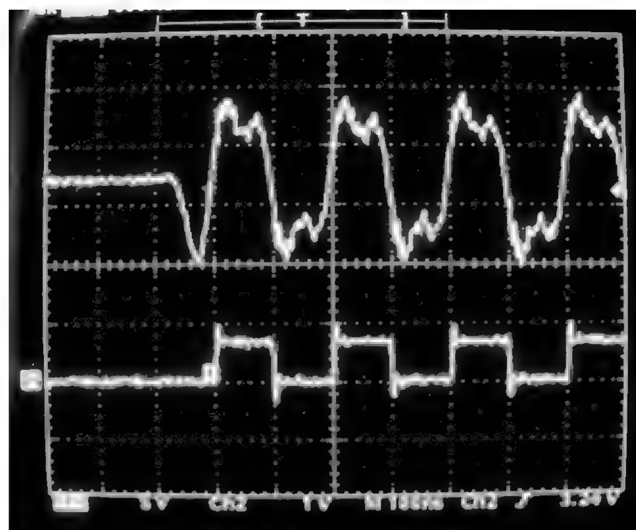
図4が受信パケット波形の測定方法です．プリアンブル部では，受信信号を差動レシーバで受けた直後のLV-TTLレベルの波形と比較し，データ部ではさらにマンチェスタ符号を復号した状態を比較してみます．写真5(a)では，プリアンブルを受信すると，きれいな5MHz方形波が出力されているようすがわかります．写真5(b)では，マンチェスタ符号を復号する回路(マンチェスタ符号のビット中央のエッジを検出してから出力が確定する)を経由した信号なので，出力が少し遅れて出ている点に注意してください．

3 フレーム送受信に関する確認

● フレームとして認識できるかどうか

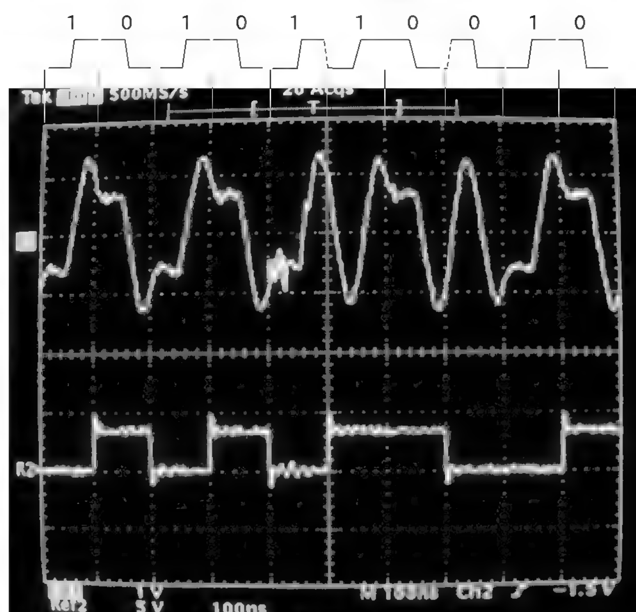
これまでの確認は信号波形が正しいかどうかを確認しただけで，これだけではパケットの送受信が正しく行えるかどうかはわかりません．実際にパケットを送信するには，正しいフレームとして認識してもらえなければなりません．そしてパケットを受信するには，フレームを正しく認識しなければなりません．オシロスコープでは観測できる範囲が狭いので，次からはロジック・アナライザを使い，フレーム送受信の全体を見てみます．

フレームの波形の確認において注意すべき点は3か所です(図5)．一つ目はフレーム情報の先頭に位置するプリアンブルを見ます．特に注目すべきはプリアンブルの最終部分に来るSFD(スタート・フレーム・デリミタ)です．プリアンブルは'1'，'0'の繰り返しをマンチェスタ符号で送出しているので，ロジック・アナライザ上は5MHzのクロック信号と同じ見えかたになります．この5MHzクロックが31パルスきた後に，マンチェスタ符号で連続した二つの'1'を示す情報がくることになって



(a) プリアンブル部

1 0 1 0 1 1 0 0 1 0



(b) データ部(マンチェスタ符号復号後)

写真5　受信パケット波形

■ column 2

間抜けな結末!?

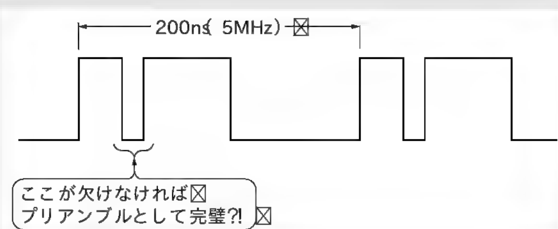
白状すると，実は当初試作した基板ではRJ-45コネクタのピンの配線を間違っていて，受信用の信号を3番と5番ピンにつないでいました．しかしその状態でも，パケットを受信するとプリアンブルらしき波形が出てくるのです．とはいえ，プリアンブルは5MHzの方形波になってほしいところが，図Bのように"H"レベルの一部が欠けたような波形になっています．

接続としては切れているにもかかわらず，波形が見えた理由を考えてみると，相手側で正常に接続されている6番ピンのUTPケーブル部分と，誤って接続した5番ピンのUTPケーブル部分がケーブル上で平行して配置されていることになり，わずかな量のコンデンサとして動作しているようになったものと考えられます(図C)．

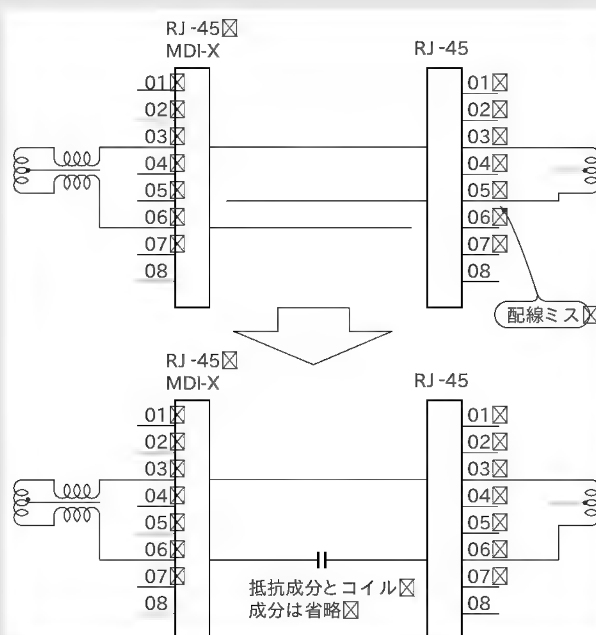
このケースで発生している現象は，電気回路の過渡現象における「不足制動」という状態になっています．これは，信号変化において状態を安定させようとする力が「不足」するため，電位が安定するまでに時間を要し，そして安定状態に達するまでの電位が大きく揺らぐ(オーバ・シュートが大きい)ことになるためです．この状態の逆の現象が「過制動」という状態で，不足制動と違い電位は暴れませんが，電位の変化が遅くなるので，基準電位の通過に余計な時間を要し，高速伝送の阻害要因になります(図D)．

表向きは，不足制動や過制動を適正制動にするためには，抵抗やコンデンサ，コイルといった要素を用いて微分方程式やラプラス変換で計算を行って対処します．しかし実際には，波形を見てから「経験と勘」で部品追加や値の変更で済ます場合も多いと思います．よく行われる対処としては，反射波が重なることによる波形の崩れには，伝送路上に抵抗(ダンピング抵抗)を入れ，反射波を押さえ込んでしまう方法がとられます．

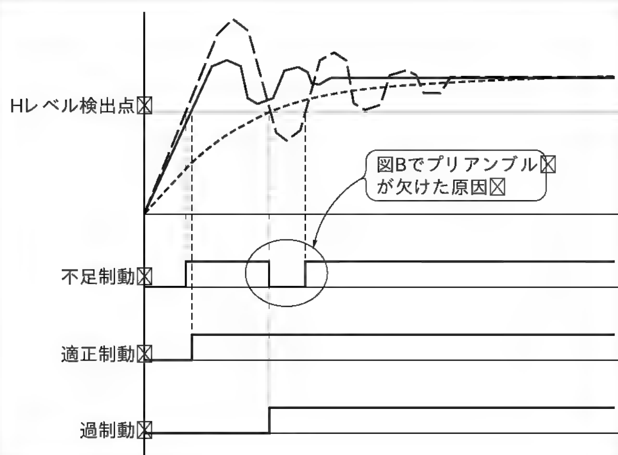
途中で配線ミスに気がつき修正したところ，プリアンブルできれいな5MHzのクロックが観測されて，ホッとしたという次第です．



図B　プリアンブルの波形



図C　RJ-45コネクタ・ピンの配線ミス



図D　過渡現象

います．見かけ上は2パルス分の20MHzクロックに見えます．このスタート・フレーム・デリミタを検出できていなければ，Ethernetのインターフェースはフレームの先頭を検出できないので，正常なパケットとみなされなくなります．

ただし，プリアンブルの先頭は多少欠けても良いことになっているので，最初からきれいな5MHzクロックが出ていなくても問題ありません．

もう一つは，フレームの最後を示す情報が明確にあるかどうかです．マンチェスタ符号でフレームの最終部分を示すための情報としては，ビット情報としてマンチェスタ符号の形態をとりません．つまり，ビット情報の中央で電位の変化がない('1'もしくはHi-Z)状態が2ビット分あるかどうかです．

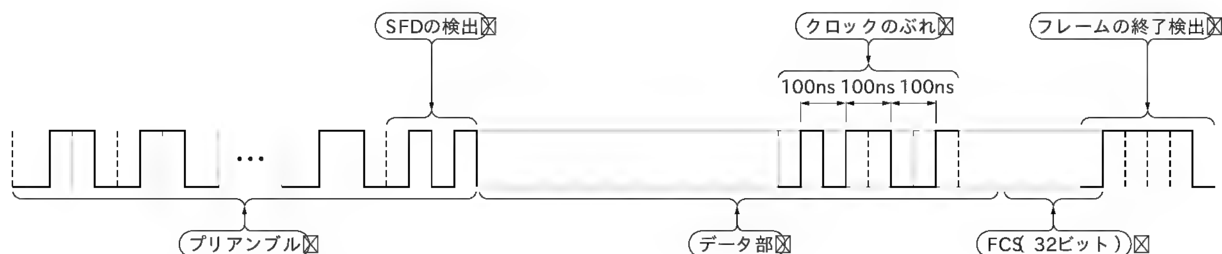


図5　フレームの波形の確認において注意すべき点

また，マンチェスタ符号のビット中央のエッジのクロック周期が10MHzになっているかどうかもチェックすべきポイントです．

図6にテスト環境を示します．パケット・キャプチャ・ツールとパケット送信テストは，同一のPCで行うことも可能です．今回使用したLANアナライザ・ソフトウェアPacMonでも，プロ版であればパケット送信機能があるので便利です．また今回はさらにNE2000を直接制御して，エラーを含んだパケットの送受信もテストしてみました．

● 送信フレームの確認

図7に，送信フレームおよび送信部の各部の動作を示します．送信部は20MHzのクロックで動作しており，プリアンブルでは62ビット分'1'と'0'を交互に出力し，最後の2ビットは'1'を連続して出力します．

次からは実際のデータ部分です．1バイトはLSBから送信してパラレル→シリアル変換しながら出力していきます．データ部が終わったら，次はFCSの出力で，CRCの32ビット分を出力します．

最後はフレームの終了を示すために，マンチェスタ符号を使わずに'1'の状態を20MHzクロックで4クロック分出力して送信は終了です．

● 受信フレームの確認

図8に，受信フレームおよび受信部の各部の動作を示します．受信部は100MHzのクロックで受信信号をサンプリングし，受信波形のエッジ検出を行っています．プリアンブル検出信号は，5MHzクロックの山と谷が内部の基準ビット列と一致した瞬間に出力されます．プリアンブル検出信号が出力されたら内部のカウンタをリセットして，90ns～110nsが経過した時点で，マンチェスタ符号のビット中央のエッジが検出できるかどうかを判定します．エッジが判定できれば，立ち上がりエッジなら受信ビットは'1'，立ち下がりエッジなら受信ビットは'0'であると判定します．また，同時にカウンタをリセットして，次のエッジ検出を待ちます．

受信アイドル状態からプリアンブル検出信号がセットされたら，受信ビットが'1'→'1'とSFDが検出されるまで待ちます．SFDが検出されたら，次のビットからが実際のデータ部分になります．

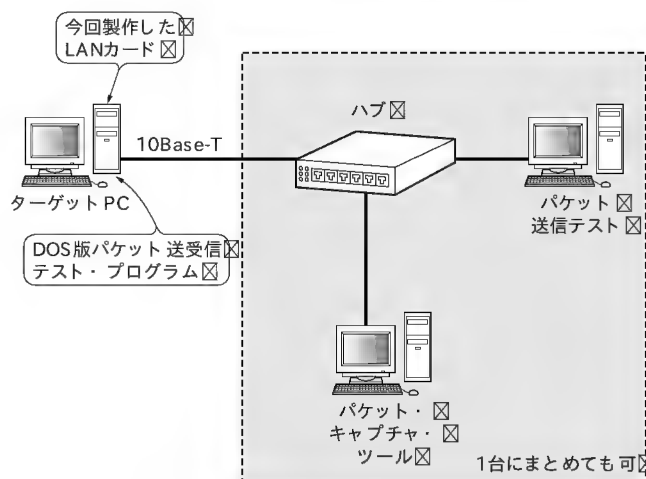


図6　フレーム確認のテスト環境

データ部では，受信ビットが確定したらシリアル→パラレル変換を行って，8ビット分そろったら1バイト受信が完了します．

フレームの終了判定は，マンチェスタ符号のビット中央のエッジが検出されなくなった時点です．ここでフレームの受信が終了します．このタイミングで，受信ビット・カウンタが0以外で半端なビット数をカウントしていたり，受信バイト・カウンタが64バイト未満，または1518バイトを超える値だった場合はフレーム・エラーとなります．またCRCが一致しなければCRCエラーと判断します．

正常なフレームであれば，受信完了ビットをセットして，PCIバス側にデータの受信を知らせます．

● 衝突検出とバックオフ動作に関する確認

さて次は，衝突の検出とそれにともなうバックオフの正常性の確認です．バックオフのための待機時間は送信トライ回数に基づく乱数によるため，確認を行うことは難しいのですが，衝突の検出についてはフレームの送出中に受信側で何かの信号を検出した場合，送信を停止し，ジャム信号を送出させ，時間を置いて再送を行うようにするもので，相手からのフレーム受信については，受信中のフレームについて，相手インターフェースにおける衝突の検出にともなう処理を検出し，受信中のフレームを破棄するというものです．

具体的には今回作成したカードを搭載したターゲットPCと，

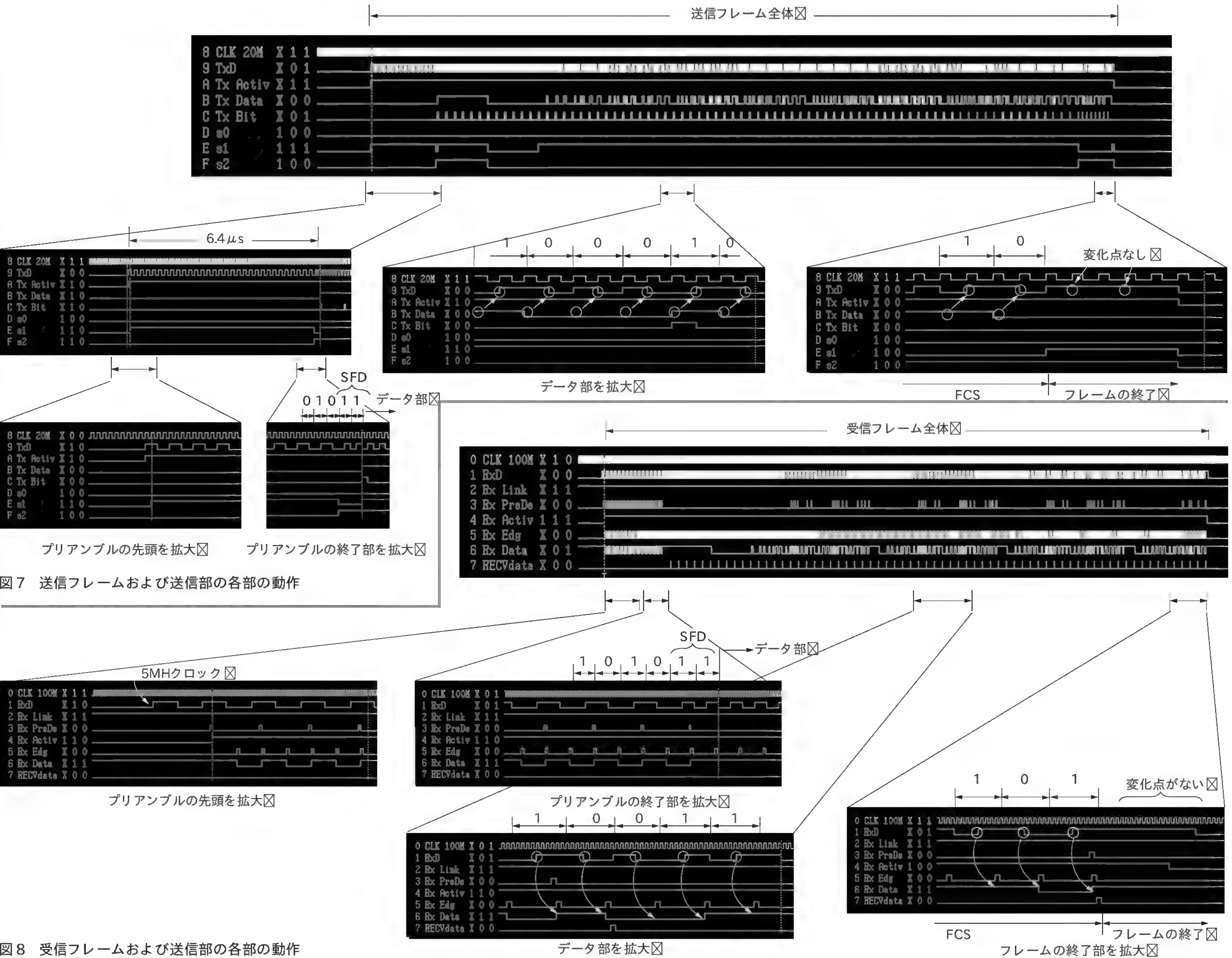


図7　送信フレームおよび送信部の各部の動作

図8　受信フレームおよび送信部の各部の動作

Information ――パーソナルメディア，T-Kernel総合サポート・センタを開設
パーソナルメディア(株)は，2月2日よりT-Kernel総合サポート・センタを開設した．同センタでは，T-Kernelのポーティングやカスタマイズ，コンサルティングなどを行う．

ターゲットPCと通信を行うPCに加え，別に2台のPCを用意します．その2台のPC間でFTPなりpingなり，適当に通信を行わせておきます．この状態でターゲットPCと通信を行うPCとの間のフレームのやりとりが正常に行われているかどうかで確認します(図9)．

なお，注意点があります．ここで使用するハブはスタック・ハブでなければなりません．スイッチング・ハブなどを使用すると，衝突が発生しないようにハブがパケットをさばいてくれてしまうので，パケット衝突の確認にはならないのです．

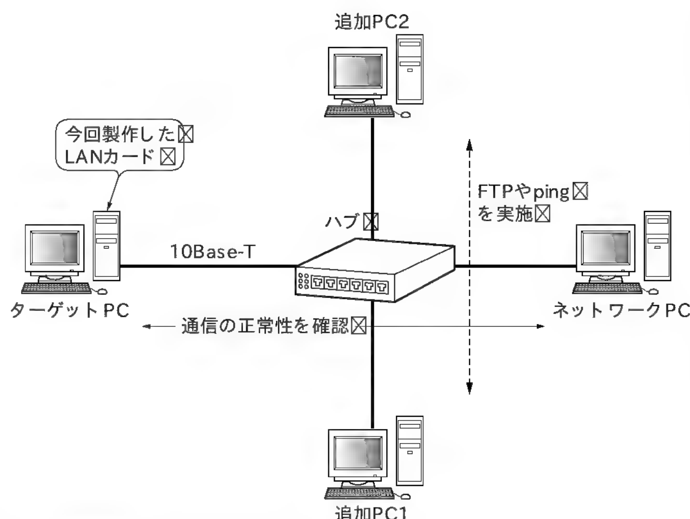


図9　衝突検出とバックオフ動作テスト

4 より実用時に近い動作確認

● いきなりメールやWWWブラウザは動かない

電気的にも問題なさそうで，単発のパケットの送受信も問題なくやりとりできることが確認できれば，次はもう少し実際の使用状態に近い形で動作確認をしてみましょう．

とはいえ，いきなりメールの送受信をしてみるとか，WWWブラウザを走らせるというわけにもいきません．これらは直接NICのレジスタをたたいて送受信を制御しているのではなく，TCP/IPプロトコル・スタックが用意しているAPIなどを使ってネットワーク通信を行うのが普通だからです．また，規模の大きなアプリケーションになると，OSのサポートも必要になってきます．

ここではもっと簡単な，ネットワークの通信テストにも使えるものとして，ping応答プログラムを用意してみることにします．

● ping応答プログラム

pingとは，IPアドレスを指定してICMPのエコー・パケットを送信し，そのIPアドレスをもった機器がそのパケットを返してくるかどうかを確認するコマンドです．LinuxやWindows環境でネットワークが使えれば，たいていの環境で実行可能なコマンドです．通信したい相手の機器(またはその途中の経路)が，生きているか死んでいるかを確認する際によく使われます．

今回作成するのは，このエコー要求パケットを受信したら，パケットのデータ部分はそのままにヘッダをエコー応答に書き換えて送り返すという処理を行うプログラムになります(リスト2)．

またICMPパケット以外に，ARPパケットの処理も必要になります．pingで指定するのはIPアドレスですが，第1章から何度も解説しているように，Ethernetフレームが扱う送信元/送信先アドレスでは，IPアドレスではなくMACアドレスを使っています．つまりICMPのエコー・パケットを送るには，まずはじめに指定されたIPアドレスを持つ機器のMACアドレスを取得する必要があるのです．

このあたりの詳しいしくみは，TCP/IPプロトコル・スタックについて詳しく解説しているほかの書籍を参考にしてください．

pingコマンドに応答するだけですが，これだけでもLANカードの動作を確認するアプリケーションとして使えます．

まとめ

物理的/電気的な送受信信号の確認から，実際に任意のパケットがデータ化けすることなくの送受信ができるかどうか，そしてもう少し実使用形態に近い動作確認として，ping応答プログラムを動かして，LinuxやWindowsマシンからpingを打ってそれに応答するかどうかの確認までができました．ここまで動けば，ハードウェアの基本的な機能には問題ないと思われます．

次章ではいよいよ，Linux用のドライバを作成してOSに組み込み，実際に各種ネットワーク・アプリケーションが動作するかを試してみます．

参考文献
1) IEEE Standerd 802.3 13，14章．
2) 泉谷 建司 著；Ethernet，(株)ソフト・リサーチ・センター．
3) Breyer&Riley 著 イデアコラボレーションズ 訳；高速Ethernetの理論と実装，ASCII
4) Ethrenetのハードを理解しよう，トランジスタ技術SPECIAL No.77，CQ出版(株)

まつもと・のぶゆき　(株)タムラ製作所
やまたけ・いちろう　来栖川電工(有)

リスト 2　ping 応答プログラム

```
/****************************************************************
    PING 応答処理プログラム for PC/AT 互換機＋ DOS
****************************************************************/

～中略～
/* PING 応答処理用 グローバル変数 */
UBYTE    MyMAC[6];                          /* 自分の MAC アドレス */
UBYTE    MyIP[4] ={192,168,0,2};            /* 自分の IP アドレス */

/* パケット・バッファ */
UBYTE    buff0[2048],buff1[2048];


/* 16 ビット値上位下位入れ替え(リトル・エンディアン環境専用) */
UWORD Word_Swap(UWORD data)
{
    return ((data<<8)&0xFF00)|((data>>8)&0xFF);
}

/* チェック・サム計算 */
UWORD calcsum(UWORD *data, int len)
{
    int  i;
    ULONG     sum;
    sum=0;
    for(i=0;i<len;i++) {
         sum=sum+(Word_Swap(*data));
         data++;
    }
    while((sum>>16)!=0){
         sum=(sum&0xffff)+( (sum>>16)&0xffff );
    }
    return( (UWORD)((~sum)&0xffff) );
}

/* PING 応答パケット作成 */
int make_IcmpPacket(UBYTE *rbuff, UBYTE *sbuff)
{
    int i,l;
    UWORD *p;
    struct Ether_Packet *rdata;
    struct IP_Header    *rip;
    struct Ether_Packet *sdata;
    struct IP_Header    *sip;
    rdata=(struct Ether_Packet *)rbuff;    /* 受信パケット・バッファ */
    rip  =(struct IP_Header *)&(rdata->data);
                                           /* 受信 IP パケット部分 */
    sdata=(struct Ether_Packet *)sbuff;    /* 送信パケット・バッファ */
    sip  =(struct IP_Header *)&(sdata->data);
                                           /* 送信 IP パケット部分 */

    /* 受信パケットの送信元 MAC アドレスを送信パケットの送信先 MAC アドレスへ
                                                              設定 */
    for(i=0;i<6;i++) sdata->dstmac[i] = rdata->srcmac[i];
    /* 自分の MAC アドレスを送信パケットの送信元 MAC アドレスへ設定 */
    for(i=0;i<6;i++) sdata->srcmac[i] = MyMAC[i];
    sdata->type    =Word_Swap(0x0800);  /* IP パケット */
    sip->header    =0x45;
    sip->tos       =0;
    sip->len       =rip->len;
    sip->no        =Word_Swap(++sno);   /* インクリメント */
    sip->off       =Word_Swap(0x0000);
    sip->ttl       =0x20;
    sip->sum       =0;                  /* 後で計算する */
    sip->protocol=1;                    /* ICMP */
    /* 送信元(自分)の IP アドレスを設定 */
    for (i=0;i<4;i++) sip->srcip[i]=MyIP[i];
    /* 送信先の IP アドレスを設定 */
    for (i=0;i<4;i++) sip->dstip[i]=rip->srcip[i];

    /* IP パケット・ヘッダ部 チェックサム計算 */
    sip->sum=Word_Swap( calcsum((UWORD *)&(sdata->data), 10) );
                                                       /* CRC 計算 */

    /* ICMP データ部分をコピー */
    l=Word_Swap(rip->len);  /* データ長 */
    for(i=0;i<( l - ( (int)sizeof(struct IP_Header)) );i++){
         sdata->data[sizeof(struct IP_Header)+i]=
         rdata->data[sizeof(struct IP_Header)+i];
    }
    /* PING コマンド用パケット設定 */
    sdata->data[sizeof(struct IP_Header)   ]=0; /* ICMP Echo */
    sdata->data[sizeof(struct IP_Header)+1 ]=0;
    sdata->data[sizeof(struct IP_Header)+2 ]=0;
    sdata->data[sizeof(struct IP_Header)+3 ]=0;

    /* ICMP パケット チェックサム計算 */
    p =(UWORD *)&(sdata->data[sizeof(struct IP_Header)+2]);
    *p=Word_Swap( calcsum( (UWORD *)&(sdata->
                    data[sizeof(struct IP_Header)]),
                    (l-sizeof(struct IP_Header))/2 )   );

    return l + sizeof(struct Ether_Header);
}

/* ARP 応答パケット生成 */
int make_ArpAckPacket(UBYTE *rbuff, UBYTE *sbuff)
{
    int i;
    struct Ether_Packet *rdata;
    struct ARP_Packet   *rarp;
    struct Ether_Packet *sdata;
    struct ARP_Packet   *sarp;
    rdata=(struct Ether_Packet *)rbuff;    /* 受信パケット・バッファ */
    rarp =(struct ARP_Packet *)&(rdata->data);
                                           /* 受信 ARP パケット部分 */
    sdata=(struct Ether_Packet *)sbuff;    /* 送信パケット・バッファ */
    sarp =(struct ARP_Packet *)&(sdata->data);
                                           /* 送信 ARP パケット部分 */
    /* 受信パケットの送信元 MAC アドレスを送信パケットの送信先 MAC アドレスへ
                                                              設定 */
    for(i=0;i<6;i++) sdata->dstmac[i] = rdata->srcmac[i];
    /* 自分の MAC アドレスを送信パケットの送信元 MAC アドレスへ設定 */
    for(i=0;i<6;i++) sdata->srcmac[i] = MyMAC[i];
    sdata->type     =Word_Swap(0x0806);    /* ARP パケット */
    sarp->hardadr   =Word_Swap(0x0001);
                          /* ハードウェア・タイプ 1(Ethernet) */
    sarp->protoadr =Word_Swap(0x0800);     /* IP */
    sarp->hardlen   =6;        /* MAC アドレスは 6 バイト */
    sarp->protolen =4;         /* IP アドレスは 4 バイト */
    sarp->code      =Word_Swap(2);         /* ARP 応答 */
    /* 送信元(自分)の MAC アドレス &IP アドレスを設定 */
    for(i=0;i<6;i++) sarp->srcmac[i]=MyMAC[i];
    for(i=0;i<4;i++) sarp->srcip[i] =MyIP[i];
    /* 送信先(ARP 要求元)の MAC アドレス &IP アドレスを設定 */
    for(i=0;i<6;i++) sarp->dstmac[i]=rdata->srcmac[i];
    for(i=0;i<4;i++) sarp->dstip[i] =rarp->srcip[i];
    return sizeof(struct Ether_Header) +
                               sizeof(struct ARP_Packet);
}

/* メイン・ルーチン */
int main(void)
{
    int  i,Send_Flg,Recv_Flg,count;
    int  packet_size0,packet_size1;
    SLONG     l;
    UBYTE     c;
    unsigned int  Org_IrqMask;
    unsigned long Org_Vector;

    struct Ether_Packet *rbuff;
    struct IP_Header    *ip;
    struct ARP_Packet   *arp;

    rbuff=(struct Ether_Packet *)buff0;    /* バッファ 0 */
    ip   =(struct IP_Header *)&(rbuff->data);
                                           /* 受信 IP パケット部分 */
    arp  =(struct ARP_Packet *)&(rbuff->data);
                                           /* 受信 ARP パケット部分 */

    /* ハイ・メモリ・アクセス・ライブラリ初期化 */
    if (_preInitHimem() != 0) {
         printf("Hi Memory LIB Initialize error¥n");
         return -1;
    }
    _maskNMI();               /* パリティ・エラー NMI 禁止 */

    printf("¥nPING answer proguram(Interface PCI NIC)¥n");

    i=PCIBIOS_Check();        /* PCI 環境チェック &PCI デバイス検索 */
    if (i<0) {
         printf("PCI BIOS error¥n");
         return i;
```

リスト 2　ping 応答プログラム

```
    }
    printf("Interfce PCI NIC device search ... VenderID %04Xh /
                          DeviceID %04Xh¥n",VenderID,DeviceID);
    l=PCI_DeviceSearch(VenderID,DeviceID,0);/* PCIデバイス検索 */
    if (l<0) {
        if (l==-1) printf("Device not found¥n");
        if (l==-2) printf("I/O space diseable¥n");
        if (l==-3) printf("IRQ not enable¥n");
        return (int)l-1;
    }
    printf("Device Found  Bus No:%d Device No:%d Function
            No:%d¥n",pciGetBus(l),pciGetDev(l),pciGetFunc(l));
    printf("Base Address %8lXh¥n",IO_Adr);
    printf("IRQ %d¥n",IRQ);

    /* ネットワーク・コントローラ初期化 */
    NIC_Init();

    /* MACアドレス取得(NICレジスタからの取得) */
    NIC_GetMACaddress(MyMAC);
    printf("My MAC address  : ");
    for(i=0;i<6;i++){
        printf("%02X ",MyMAC[i]);
    }
    printf("¥n");
    /* 自分のIPアドレスを表示 */
    printf("My IP address   : %d.%d.%d.%d¥n",
                         MyIP[0],MyIP[1],MyIP[2],MyIP[3]);

    /* NIC割り込み初期化 */
    NIC_Interrupt_Init();

    /* 割り込みベクタ初期化 */
    Org_IrqMask=_maskIRQ(IRQ,1);
                    /* マスク状態を保存し、割り込み禁止に設定 */
    Org_Vector=_hookIRQ(IRQ,&NIC_Interrupt);
                    /* ベクタを保存し割り込み処理関数を設定 */
    _maskIRQ(IRQ,0);    /* システム・コントローラ割り込み許可 */

    /* 送信部イネーブル/ディセーブル制御 */
    NIC_SendCtrl(1);    /* イネーブル */
    /* 受信部イネーブル/ディセーブル制御 */
    NIC_RecvCtrl(1);    /* イネーブル */

    /* その他の初期化処理 */
    while(kbhit())getch();  /* キー・バッファ・クリア */
    count=0;
    sno=0;
    Send_Flg=0;
    Recv_Flg=0;

    printf("PING Ack Ready!¥n");

    /* メイン・ループ */
    while(1){

        if (Packet_RecvReady) { /* 受信パケットあり! */
            printf("Recv Packet (%d)¥n",++count);
            packet_size0=NIC_RecvPacket(buff0);
                                  /* パケット取り出し */

            /* 受信パケット種別判定 */
            if (rbuff->type=Word_Swap(0x0800)) {
                                  /* IPパケット? */
                if (checkAdr(MyIP, ip->dstip, 4)) {
                                  /* 自分宛てのIPアドレスか? */
                    if (
                        (ip->protocol = 1)
                                  /* プロトコル:ICMP */
                     &&
                        (rbuff->data[sizeof(
                            struct IP_Header)] = 8)
                                          /* エコー要求 */
                    ) {
                        printf("IP Packet / ICMP /
                                          Echo Req ¥n");

                    packet_size1=make_IcmpPacket(buff0,buff1);
                                  /* PING応答パケット作成 */
                        i=NIC_SendPacket(buff1,
                                          packet_size1);
                                  /* パケット送信 */
                        if (i) printf("Send Packet Write
                                    Error!¥n");
                        else {
                            printf("Send Packet Write ok!
                                      (%d)¥n",sno);
                            Send_Flg=1;
                        }
                    }
                }
            }
            if (rbuff->type=Word_Swap(0x0806)) {
                                          /* ARPパケット? */
                if (arp->hardadr=Word_Swap(0x0001)) {
                            /* ハードウェア・タイプ 1(Ethernet)? */
                    if (checkAdr(MyIP, arp->dstip, 4)) {
                                  /* 自分宛てのIPアドレスか? */
                        printf("ARP Packet¥n");
                packet_size1=make_ArpAckPacket(buff0, buff1);
                                          /* ARP応答パケット作成 */
                        i=NIC_SendPacket(buff1,
                                          packet_size1);
                                          /* パケット送信 */
                        if (i) printf("Send Packet Write
                                          Error!¥n");
                        else {
                            printf("Send Packet Write ok!
                                              ¥n");
                            Send_Flg=1;
                        }
                    }
                }
            }

    /*  buffer_dump(buff0,packet_size0); */

        }

        if (RecvError) {    /* 受信系エラーあり */
            if (Packet_RecvReady) break;
                /* 受信完了バッファがあれば、先にそちらを処理する */

            if (RecvError&RECVx_CRCErr)
                    printf("Recv Packet CRC error!¥n");
            if (RecvError&RECVx_FrameErr)
                    printf("Recv Packet Frame error!¥n");
            if (RecvError&RECVx_BuffOver)
                    printf("Recv Packet Buffer Overflow!¥n");
            NIC_RecvErrorRecovery();/* 受信系エラー復帰処理 */
        }



        if (SendError) {        /* 送信系エラーあり */
            printf("Send error!  Send retry¥n");
            NIC_SendRetry();    /* 送信バッファ再送 */
        }

        if (Send_Flg) {         /* 送信中 */
            i=NIC_SendBufferCheck();/* 送信完了確認 */
            if (i==0) {
                printf("Send Packet Success¥n");
                Send_Flg=0;
            }
        }

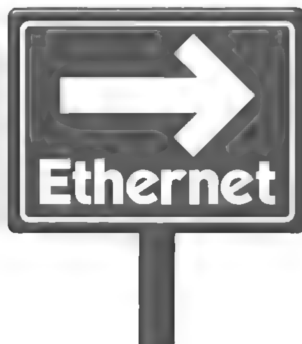
        if (kbhit()) { /* キーボード入力チェック */
            c=getch();
            if ((c=='Q')||(c=='q')) break;    /* そのまま終了 */
            if ((c=='E')||(c=='e')) break;    /* そのまま終了 */
        }

    }

    /* 割り込みベクタを戻す */
    _freeIRQ(IRQ,Org_Vector);     /* ベクタを元に戻す */
    _maskIRQ(IRQ,Org_IrqMask);    /* 割り込みマスクの状態を元に戻す */

}
```

Appendix



Ethernetケーブルで電源を供給する

Power over Ethernetの概要

松本 信幸

● Ethernetインターフェースによる電力供給

インターネット電話のように，家庭用電力源とは別の電力供給があることが望ましいものや，無線LANのアクセス・ポイントのような，天井裏などにコンセントがないといった，もともと配線が困難な場所へ設置を行う可能性があるような機器においては，信号伝送を行うケーブルを用いた電力供給が望まれることがあります．

また，家庭内で使用する機器のように小型化された省電力機器が多数存在すると，コンセントの周りには山のようにACアダプタが置かれて余計な場所を取ることもあります．これらを回避するために，Ethernetの勧告では，IEEE802.3afとして，UTP（Unshielded Twisted Pair wire）ケーブルを用いた電力の伝送が規定されています．

● IEEE802.3afとは

IEEE802.3afは，一般に「Power over Ethernet」（PoE）と呼ばれるものですが，その内容はEthernetの中でも特にUTPケーブルを用いることに特化したもので，CSMA/CD方式をはじめとする従来までの各種勧告との間に因果関係や影響はほとんどありません．

したがって，IEEE802.3afが関係してくるインターフェースは10/100Base-TXと1000Base-Tだけであり，その動作にも大きな影響はありません．光ファイバ・ケーブルを用いるものや同軸ケーブルを用いる1000Base-LX/SX/CXなどは対象外となります．

IEEE802.3afの目的は一つで，UTPケーブルを用いて接続される機器に対する電力の供給を行うことだけです．しかし，ここで問題となるのが，電力用に用いるピン・アサインです．Ethernetに用いるUTPケーブルのコネクタはRJ-45で，8本のケーブル接続をもっています．10/100Base-TXでは，このうちの1，2，3，6の四本が送受信ペアとして使用され，4，5，7，8の2ペア（ペア1とペア4）が未使用となっています．普通に考えれば，この未使用のペアを用いてしまえばよいのですが，1000Base-Tの場合，10/100Base-TXで未使用な4，5，7，8番ピンも使用することになっていますし，安価なUTPケーブルには1，2，3，6番ピンしか接続されていないものもあります（それだけで十分だと記載している古い書籍もある）．

このため，IEEE802.3afでは，1，2，3，6番ピンを用いて電力供給を行う方法と，4，5，7，8番ピンを用いる方法の両方を規定しています．また，信号ピンである1，2，3，6番ピンを使用する場合には，MDIとMDI-Xの両方について規定しています（表A）．

表A　IEEE802.3afによる電源供給ピン配置

ピン番号	直流電力供給			交流電力供給	
	Aタイプ（信号多重）		Bタイプ（空きピン使用）	モードA	モードB
	MDI-X	MDI			
1	マイナス	プラス		プラス/マイナス	
2					
3	プラス	マイナス		マイナス/プラス	
4			プラス		プラス/マイナス
5					
6	プラス	マイナス		マイナス/プラス	
7			マイナス		マイナス/プラス
8					

Power over Ethernetを利用したネットワークを構成する場合，「どこから電力を供給するか」という問題があります．IEEE802.3afでは，電力供給用の機能をPower Sourcing Equipment（PSE）と呼び，受電側の機能をPowered Device（PD）と呼んでいますが，要はこのPSEが，どこに配置されるかということです．一つにはハブのように通信を行う相手どうしにおいて電力の供給も行う（Endpoint PSE）方式であり，もう一つは情報の送受信を行う機器の間に電力を供給する装置を中継する（Midspan PSE）方式です（図A）．

● データ多重の電力伝送：タイプA

タイプA（IEEE802.3af上の記述は「Alternative A」）として規定されている方式は，データ送受信ペアである1，2，3，6番ピンを用いた電力伝送です．ここで用いる方法は，1，2番ピンの送信ペアと3，6番ピンの受信ペアを一組のペアとして用いる方法です（図B）．

電力は電圧（電位差）と電流の積です．さて1，2番ピンと3，6番ピンのそれぞれの電位差は，わずかに2V程度しかありません．これで装置を駆動するために必要な数Wの電力を供給しようとすると1A以上の電流を必要とします．

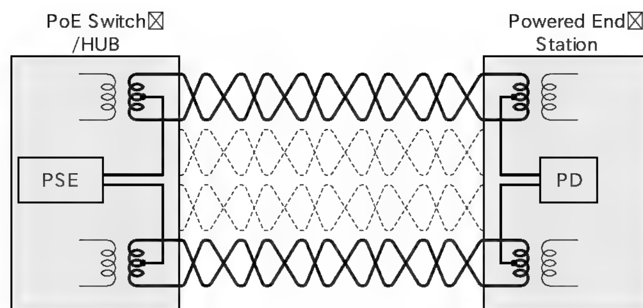
Ethernetに限らず，多くの高速伝送用インターフェースではトランスやコンデンサによって電流成分を遮断するようになっています．そのため情報の伝送に用いられる1，2番ピンを流れる電流i_1はそのまま送信元に戻って行くことになります．同様に3，6番ピンを用いて送信元からやってくる電流も戻っていきます．さて，ここで供給されるべき電力用の電流Iですが，これは1，2番のペアと3，6番のペアで電流ループを作ることになります（キルヒホッフの法則を参考にしてほしい）．

情報の伝送に必要な電位差は，あくまで1番ピンと2番ピンの間に2V程度あればよく，3，6番ピンについても同じです．1，2番ピンの中間電位と3，6番ピンの中間電位の間に，どれほどの電位差が生じていようとも，通信には基本的に影響ありません．したがって，この送信ペアと受信ペアそれぞれの中間電位による電位差を利用して電力の伝達を行えばよいわけです（図C）．

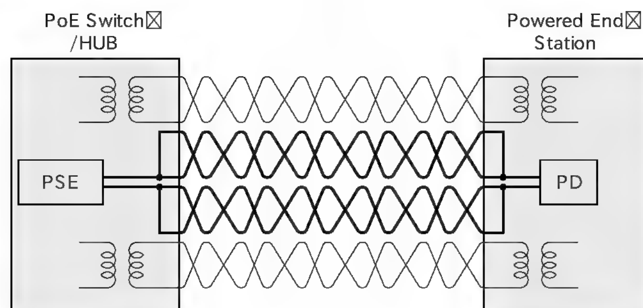
● 空きピン利用の電力伝送：タイプB

もう一つの方法は，10/100Base-TXでは未使用の4，5番ピンと7，8番ピンのペア1とペア4を利用するタイプB（「Alternative B」）です．

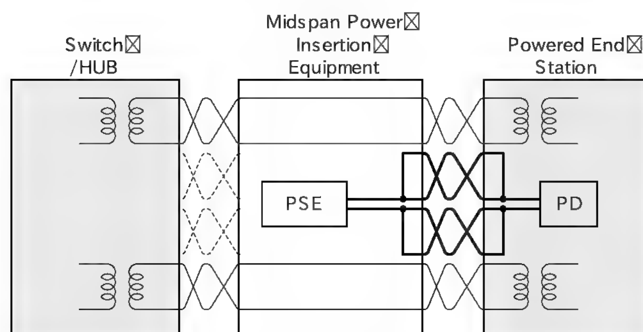
IEEE802.3afでは，未使用ピンとして4，5番ピンと7，8番ピンを使用する場合，それぞれ短絡してしまうように書かれています．しかし，この接続では，RJ-11の電話線をADSLなどとまちがえたことによる誤接続が行われると，L1とL2（つまり4番ピンと5番ピン）の



(a) タイプA Endpoint PSE方式

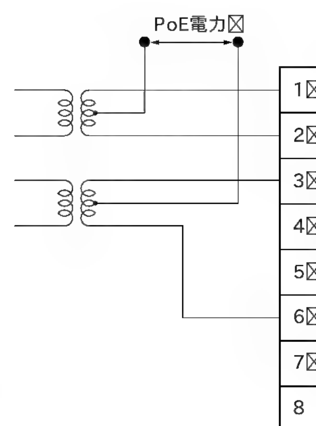


(b) タイプB Endpoint PSE方式

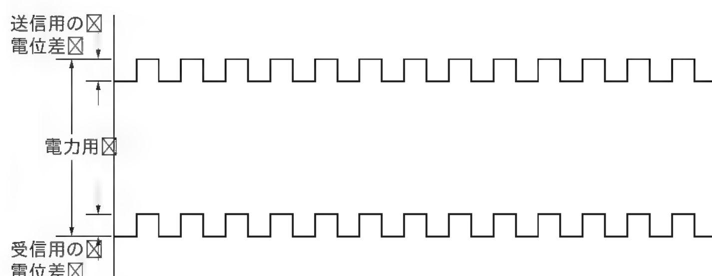


(c) タイプB Midspan PSE方式

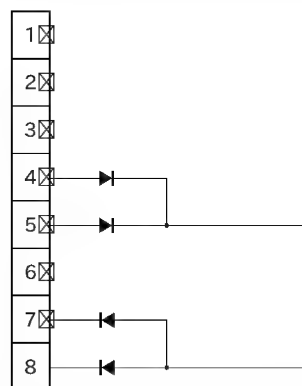
図A IEEE802.3afによる電源供給回路



図B
データ多重の電力伝送方式の
電力取り出し部



図C データ多重の電力伝送方式の送受信電圧と電力の関係



図D 空きピン利用の電力伝送方式の保護ダイオードの入れ方

ショートが発生するので，個人的にはあまりお勧めしません．勧告には書かれていませんが，(気休め程度だが)整流ダイオードでも入れておいたほうがよいかもしれません(図D)．

Endpoint PSE方式の場合であれば，タイプA/B両方とも実現可能ですが，Midspan PSE方式はこのタイプBが用いられます．Midspanで信号多重であるタイプAを実現しようとすると，パルス・トランスで一度終端しなければならないためです．このため，市販のIEEE802.3af対応の無線アクセス・ポイントに付属されている電力供給用のモジュール(Midspan Power Insertion Equipment)などは，タイプBの空きピン使用の供給方式となります．ちなみに，ものによっては4番ピン，5番ピンのいずれか一方と，7番ピン，8番ピンのいずれか一方のみを使用して，4，5番ピン，7，8番ピンが接続されていないモジュールも見受けられるので注意が必要です．

● 電力源としてのPoE

(多少乱暴な見かただが)通信装置などで供給される電力には2通りがあります．これはAC電源とバッテリなどという意味ではありません．一つはコンセントから供給される家庭用AC電源などの「供給される電力」と，装置が実際に動作するために必要となる「動作のための電力」です．ターミナル・アダプタなどの多くの装置では，ACアダプタを用いていますが，このコンセント側である1次側が「供給される電力」であり，5Vなど，動作電圧に変換された2次側が「動作のための電力」です．

一般には，「動作のための電力」に「1次側から2次側への変換の際に消費される電力」を加えたものが，その装置の消費電力として扱われます．あまり知られていませんが，装置全体の消費電力のうち，約2割程度(少ないものでも15%程度)は変換の際に消費されています．

さて，Power over Ethernetとして供給される電力についてですが，ここで供給される電力は1次側電力です．そのままで装置が使用する+5Vや3.3Vような電圧で供給されません．ですからACアダプタで+5Vを供給されるような装置であっても，Power over Ethernet

表B　IEEE802.3afの電源供給能力(直流)

	パラメータ	単位	最小値	最大値
1	出力電圧(直流)	V_{dc}	44	57
2	負荷電圧	V	44	57
3	負荷電力	W	0.44	15.4
4	最大出力電流	mA	350	

表C　IEEE802.3afの消費電力によるクラス

クラス	扱い	最小出力
0	デフォルト	15.4W
1	オプション	4.0W
2	オプション	7.0W
3	オプション	15.4W
4	リザーブ	

を使用する際には装置内に電源回路が必要となります．

IEEE802.3afで規定されている電力伝送の方法ですが，これには直流方式と交流方式の二つがあります．よく見かけるものは直流方式のものですが，受電側である装置では，IEEE802.3afサポートとうたうためには直流方式と交流方式の両方で受けることが可能でなければなりません(整流用のダイオード・ブリッジを入れておく程度)．交流の周波数は，商用電源の50/60Hzと異なり，500Hzが用いられています．

電力源としてもっとも問題となるのは，その供給能力です．接続して使用できるパワー・デバイスの消費電力の上限は15.4Wからと，けっこう大きめの装置でも駆動することが可能となっています(表B)．また，IEEE802.3afでは，消費電力によってクラス分けがなされています．ただし，これはオプションで，デフォルトは最大と同じ15.4Wです(表C)．

● 筆者の個人的意見(？)

IEEE802.3afとして電源系の作業を行った際に気になったのは,「いつまで電話の呪縛が続くのであろうか…」でした．というのも，－48Vは電話交換機が用いる電圧源のものだからです．電話局にある局用交換機では，12Vバッテリを4台直列に接続し，52V程度の電源を用いて運用します．こうすると，通常時はそれぞれのバッテリに13Vかかることになり，充電状態となります．そして，商用電源がダウンすると，バッテリから48V(12V×4)を供給して一時的にしのいで，その間に自家発電を起動するようになっています．

確かに，ある程度の長さのケーブルを使用して，それなりの電力を伝送するためには，いい加減な電圧では，供給先で使用できなくなっている可能性もあります．

端末は，いかに安価に作るかが重要となります．3端子レギュレータを用いて簡単な電源回路を組み込むのであれば，供給電圧は24Vであったほうがありがたい…というのが感想です．

現実問題として，2次側に用いる＋5Vや＋3.3Vを生成する3端子レギュレータの入力は24V程度までとなっています．24Vを出力するものでも入力は30V程度までで，3端子レギュレータを用いる電源回路では，トランスで18V程度に変圧してから整流し，3端子レギュレータに供給しています．

－48Vから＋5Vを生成するDC-DCコンバータ(オンボード電源)などは，局用の交換機を作っている会社の実験室などに行くとけっこう落ちている(？)ものなのですが，あまり市販はされていないのでめんどうです．

電源回路を簡単に作るには，交流電圧で入力されたものをトランスで18V程度に変圧するのがいちばん楽なので，IEEE802.3afでも，交流を用いたEndpoint PSE方式であればありがたいものです．もっとも交流の電源を用いると，いろいろと認定などが絡んでくるので，回路設計以外の部分がめんどうになるのですが….

● 市販の装置における使われ方

Power over Ethernetの実装が急激に増える製品群の一つに，無線アクセス・ポイントが挙げられます．これらの製品は，ACアダプタとPoEの両方から電力の供給ができるようになっているものが多くあります．

ここで問題となるのが，2系統ある異なった電源をどのように配置するかということです．一つは，よく見かける製品同様，2次側電源としてACアダプタから＋5V程度を供給してもらい，PoEは装置内に－48Vから＋5Vや＋3.3Vを生成するための電源回路を用意する方式です．もう一つの方法は，内部にPoEから供給される－48Vを元に＋5Vや＋3.3Vを生成する回路はそのままで，ACアダプタからも48Vを供給するようにして，共通化する方式です．どちらが良いかは電源回路部の信頼性とコストのトレード・オフになるでしょう．

前者のほうがコストとしては高くなるでしょうが，電源系が2次側の供給点まで二重化されるので，電源部分の信頼性は後者に比べて高くなります．後者の場合，いずれの経路で電力が供給されても2次側を生成するのは共通部分なので，ここがダウンすると装置は停止してしまいます．開発を行う際には，ターゲットとなる市場を十分に検討する必要があるでしょう．

＋5Vなどの生成部はともかく，IEEE802.3af準拠とした製品のRJ-45付近の回路としては，直流方式にも交流方式にも対応できるように整流用のダイオード・ブリッジが入り，さらにタイプA(1，2，3，6番ピン型)とタイプB(4，5，7，8番ピン型)の両方が使用できるようになっています．勧告上選択肢が多いのは，ある意味ではありがたいのですが，すべてサポートしないと「準拠」といえないのもめんどうなものです．

● PoEはノイズに注意

PoEを利用する場合，Ethernetで用いるUTPケーブルの先に電源回路がきます．先述のとおり，－48Vから．実際に使用する＋5Vや3.3Vを生成しなければならず，実はここで一つの問題が生じてきます．それは電源回路としてよく用いるDC-DCコンバータのスイッチング部によって発生するノイズがUTPケーブルに逆流し，UTPケーブル自身がアンテナとなり電波を放出することがありえるのです．PoEを使用する場合，VCCI(Voluntary Control Council for Interference by Information Technology；情報処理装置等電波障害自主規制協議会)などの測定は慎重に行う必要があります．

まつもと・のぶゆき　(株)タムラ製作所

第**5**章



設計したオリジナルLANカードを実際に活用するために

Linux用デバイス・ドライバの作成法

山際 伸一

ハードウェアが完成しても，それを制御するソフトウェアがなければネットワークに接続することはできない．ここでは設計/製作したオリジナル仕様のLANカードをLinux上から実際に活用するために，Linux用のデバイス・ドライバを作成し，動作の確認を行うところまでを解説する．（編集部）

はじめに

サーバやワークステーションのOSとしてではなく，組み込みOSとしてもLinuxの価値が高まっています．組み込みでLinuxを使うためにはEthernetを主とするネットワークをもつターゲット・ボードで，ネットワーク・デバイス・ドライバを構築できることがシステムを開発するうえでの重要なポイントのひとつです．そこで，本章では，このような新たなデバイスを追加する際のネットワーク・デバイス・ドライバの構築方法を解説します．

ここでは，現在もっとも広く用いられていると考えられる2.4系カーネルでのネットワーク・デバイス・ドライバの構築方法を説明します．Linuxでのネットワーク・デバイス・ドライバは，OSのどの部分でどのような処理を行っているかに加え，実際のドライバの内部構造についても触れます．

また，Linux上でネットワーク・デバイス・ドライバを構築するうえで大事な概念であるソケット・バッファについても解説します．ソケット・バッファの恩恵を受けて，Linuxでは，ドライバと上位のプロトコル・レイヤとの間で，きれいにデータの受け渡しが実現できています．

その後，ネットワーク・デバイス・ドライバを第3章で設計したオリジナル仕様LANカード(IF_NIC)の仕様に基づき，実際に構築していきます．

1 Linuxネットワーク・デバイス・ドライバ入門

● ネットワーク・デバイス・ドライバの位置付け

Linuxでは，図1に示すようなネットワーク用のプロトコル・スタックがあり，ネットワーク・デバイス・ドライバはソフトウェアの最下層に位置しています．ネットワーク・デバイス・ドライバは上位のプロトコルからのデータ，またはネットワーク・デバイスからのデータをその間で受け渡しする役目を担っているわけです．

図1に示すように，ネットワーク・プロトコルには複数のものがあります．これらを吸収し，おのおのの通信仕様に従って，ネットワーク・デバイスに指令を出すということもネットワーク・デバイス・ドライバの役目です．本章ではTCP/IPを目標とするEthernet互換のデバイスがターゲットになっているわけですから，ドライバはIEEE802仕様に準拠するデータをやりとりするわけです．これはつまり，データの長さやフォーマット，送信タイミングなどをIEEE802の仕様に従って行わなければならず，それに外れるような場合，エラーとして検出する必要があるということです．

Linuxでは，当該ドライバがEthernet互換であるということを事前に宣言することにより，IPレイヤとドライバの間で自動的にEthernetパケットを作ってくれるレイヤが存在します．これにより，ドライバ内で処理しなくてはならない仕事を減らすことが可能になります．反対に，受信の際にもEthernetフレームをそのままドライバの上位に渡すことにより，IPレイヤの手

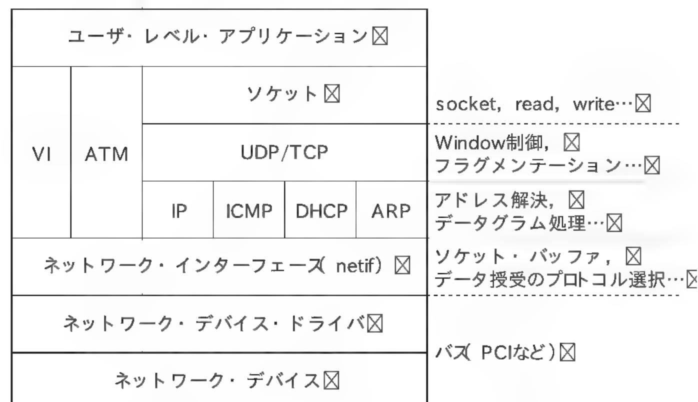


図1 Linuxにおけるネットワーク用プロトコル・スタック

前でIPパケットのみを取り出してくれます．この機構には，後で説明するソケット・バッファが大いに役立っています．

ここで，Ethernet向けのLinuxデバイス・ドライバがどのような方法でデータをやりとりするかを考えてみましょう．

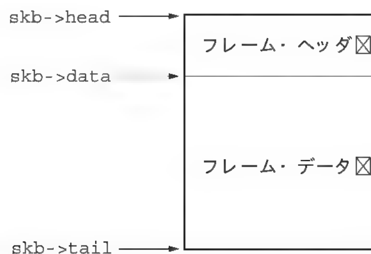
まず，送信について考えてみます．送信する際には，IPレイヤからEthernetのフレームが作られてきます．ドライバはデバイス上の送信バッファに空きがあるか否かを調べ，空きがない場合，そのデータはあとで処理することをマークします．空きがある場合，フレームをデバイスにコピーし，送信要求をデバイスに対して出します．一般的なネットワーク・デバイスは送信完了，またはエラーのときにドライバへの割り込みを発生させます．

一方，受信の際には，デバイス上にフレームが受信されたとすると，デバイスによりドライバへの割り込みが発生します(割り込みを発生しないものもあるかもしれないが，一般的なネットワーク・デバイスなら発生する)．ドライバはその割り込みにより受信フラグをチェックし，デバイスからフレームをコピーします．この際に，ドライバ内部には事前にフレームを受信するためのバッファが確保されていることに注意してください．コピーのあと，そのフレームを上位のプロトコルへと渡し，受信が完了します．

● ソケット・バッファ

上位のプロトコルから渡される，または上位に渡すバッファには，Linux独自のフォーマットがあります．このバッファのことをソケット・バッファ(Socket Buffer)と呼びます．ソケット・バッファは構造体で，データ部分と属性を保持するメンバからなります．ドライバで行うデータ・フローはこのソケット・バッファとデバイス間でのデータ移動が主たる仕事になります．

図2に，ソケット・バッファの構造とそれを制御するための代表的な関数を示します．Linuxのソースを眺めていると，skbというポインタをよく見かけることでしょう．これがソケット・バッファです．ソケット・バッファは構造体であり，それが管理するデータの特徴を保持しています．たとえば，保持しているデータはEthernetであるとか，データ部分のバイト数などです．



▶ skb_reserve (struct sk_buff *skb,unsigned int len)☒
ヘッダ領域を作る関数．data ポインタとtailポインタを同時にlen分だけ増加させる．☒
▶ skb_put (struct sk_buff *skb,unsigned int len)
データ領域を作る関数．tail ポインタをlen分だけ増加させる．☒
▶ struct sk_buff *skb_pad (struct sk_buff *skb,int pad)
pad バイト分0でパディングする．☒

図2 ソケット・バッファの構造

ソケット・バッファを確保するにはdev_alloc_skbカーネル・コールに作成するデータ用のバッファ領域の大きさを与えます．この際に作成された領域は，デバイスからのフレームを保持するための領域となります．すなわち，上位プロトコル層およびデバイスから受け取ったデータは，この領域を介してやりとりされます．開放する場合は，dev_kfree_skbカーネル・コールを呼びます．

ソケット・バッファの保持する情報で，ドライバにとってもっとも重要なものは，ヘッダ・ポインタ(head)，データ・ポインタ(data)，テイル・ポインタ(tail)です．ヘッダ・ポインタは，ソケット・バッファが保持するフレーム・ヘッダの先頭に当たります．

データとは，フレームのデータの先頭を指しています．テイル(tail)とは，ソケット・バッファが保持するフレームの最後を指しています．

これらのポインタを操作することで，送信する，または受信したデータの長さを指定できる，シンプルなインターフェースが実現されています．

ソケット・バッファが確保された直後は，ヘッダ，データ，テイルのポインタはすべてバッファの先頭を指しています．

ソケット・バッファを操作する際に用いる二つの代表的なカーネル・コールがあります．skb_reserveカーネル・コールはデータとテイル・ポインタを同時にずらします．これは，ヘッダ部分のアラインをするときに使います．skb_putカーネル・コールは，テイル・ポインタを与えられたバイト数分ずらします．つまり，フレーム内のデータ・パケット部分を作り出します．

デバイスの機能によっては，最小フレーム・サイズが定義されている場合があります．このような場合，0でパディングをしてくれるソケット・バッファ操作カーネル・コールskb_padがあります．これでパディングする際には，ソケット・バッファのデータ部が伸張され，送信しようとしているデータの後に指定したバイト分0が追加されます．

送信の際には，後述する送信用ドライバ・メソッドにソケット・バッファのポインタが渡ってきます．それを処理して，デバイスにフレームとして送信するわけです．一方，受信では，受け取ったデータをソケット・バッファに移し，netif_rxカーネル・コールでソケット・バッファを上位プロトコル・レイヤに渡します．

ソケット・バッファで重要なメンバは，devとprotocolです．devには，それを操作するドライバを登録し，protocolにはEthernetなどのデータのタイプを指定します．

● ネットワーク・デバイス・ドライバの構造

本章では，ドライバをモジュール化し，カーネルに組み込むことを前提として説明していきます．

Linuxネットワーク・デバイス・ドライバを記述していくうえで，イベントの処理順序を考えておくと理解が進みます．そこで，Linuxネットワーク・デバイス・ドライバの構造をほかのレイヤとの入出力を元に，モジュールのロードから，通信，モジュールの取り外しの一連の処理に必要な関数を説明していきます．

▶ドライバ・モジュールのロード

ドライバの生命は，ドライバ・モジュールのロードから始まります．insmodコマンドがスーパ・ユーザ権限でコマンド・ラインから実行されると，ドライバ・モジュールのinit_module関数が呼ばれます．Linuxネットワーク・デバイス・ドライバは，ほかのキャラクタ・デバイス・ドライバのように，モジュールの初期化（init_module）がドライバのエントリ・ポイントとなり，ドライバの登録と初期化を行います．

init_module関数では，次の手順を実行する必要があります．

（1）デバイスの認識

PCIバスを介したネットワーク・デバイスの場合，ベンダID，デバイスIDなどからデバイスを発見する作業を記述する必要があります．デバイスを発見できた場合には，デバイスのメモリをドライバにマップし，直接アクセスできるようにする必要があります．

（2）request_irqカーネル・コールで割り込みを登録

ネットワーク・デバイスは通常，送受信の完了などの通知に割り込みを使うので，ここで割り込みハンドラをIRQに一致させて，カーネルに登録する作業が必要になります．割り込みハンドラは，

```
static void interrupt_handler(int irq,
void *dev_instance, struct pt_regs *regs);
```

のようなプロトタイプをもつ関数であり，割り込み発生時に呼ばれます．staticで定義することに注意してください．これは，ほかのドライバとシンボルの重複を避けるためです．

（3）Ethernet用のnet_device構造体を作る

Linuxでのネットワーク・ドライバでは，net_device構造体をカーネルに登録することがすべての作業であるといえます．net_device構造体は/usr/src/linux/include/linux/netdevice.hで定義されています．

ネットワーク・デバイス向けのnet_device構造体を作成する際には，alloc_etherdevカーネル・コールを使います．このカーネル・コールは，net_device構造体をkmallocにより配置し，ネットワーク・デバイス向けに初期化する関数です．

次に，アロケートされたnet_device構造体にドライバ固有の情報を登録する必要があります．net_device構造体の中の次のメンバを初期化してください．

● name

これはethなどのカーネルが扱うネットワーク・インターフェースの名前です．ifconfigコマンドで見えるeth0，loなどの名前は，ここで決定されています．初期化しないとデフォルトの"eth%d"になります．

● watchdog_timeo

タイムアウトする時間を設定します．HZマクロを使うと，秒単位の時刻を容易に指定することができます．

● dev_addr

MACアドレスを設定します．これに関しては，リトル・エンディアンであることに注意してください．

最後に，ドライバが上位プロトコルとのインターフェースを提供するメソッド群をnet_device構造体に登録する必要があります．

表1が，本章のドライバが扱うメソッド群です．これらは最低限のドライバが提供しなければならないインターフェースとなります．メソッドはstaticで定義されることに注意してください．ほかのドライバとシンボルの重複を避けるためです．

表1を見て「受信処理がないぞ？」と疑問に思うかもしれません．受信処理に関しては，割り込み駆動で行われるので，割り込みハンドラの仕事になります．したがって，受信処理の詳細に関しては，関数のプロトタイプなどの仕様はありません．

また，上位プロトコル・レイヤとの間のフロー制御を開始・停止の指示を出してやる必要があります．このフロー制御を開始する，つまり送信データの受け付けをドライバが許可するにはnetif_start_queueカーネル・コールを呼び，送信データの受け付けを禁止する際にはnetif_stop_queueカーネル・コールを呼びます．

（4）ネットワーク・デバイス構造体を登録

net_device構造体を初期化し，上位プロトコル・レイヤへ

表1 メソッド・プロトタイプとその説明

int open(struct net_device *dev);
ifconfigコマンドでupされた際にこのメソッドが呼ばれる．送信用/受信用のソケット・バッファを作成し，さらにデバイスをアクティブにし，netif_start_queueカーネル・コールで上位プロトコル・レイヤの送受信処理を開始する．
int stop(struct net_device *dev);
ifconfigコマンドでdownされた際にこのメソッドが呼ばれる．netif_stop_queueカーネル・コールで上位プロトコル・レイヤの送受信処理とデバイスを停止させて，ソケット・バッファなど，メモリ・リソースを解放する．
int start_xmit(struct sk_buff *skb, struct net_device *dev);
このメソッドはIPレイヤなどの上位のプロトコル・レイヤから，送信すべきデータがある際に暗に呼ばれ，送信処理を行う．上位プロトコルから渡されるソケット・バッファを解析し，データ部分をデバイスに書き込む．
struct device_stats *eth_get_stats(struct net_device *dev);
ifconfigコマンドにより表示される情報（たとえば，送受信データ量，エラーなど）を返すメソッド．このメソッドはnet_device_stats構造体の情報をアップデートすることがおもな仕事になる．
void tx_timeout(struct net_device *dev);
データの送信をデバイスに要求した後に，net_device構造体のwatchdog_timeoメンバに従った時間で，その完了が確認されないときは送信タイムアウトが発生する．その際にどのような処理を行うかをこのメソッドで決定する．

のメソッドを登録したところで，このnet_device構造体をカーネルに登録します．このとき，register_netdevカーネル・コールを使います．この関数を実行した後，ネットワーク・インターフェース名がカーネル内に登録され，upする準備が整えられます．

▶ドライバ・モジュールのアンロード

ドライバ・モジュールがアンロードされる際にはcleanup_module関数が呼ばれます．

cleanup_module関数では，デバイスの登録解除と割り込みルーチンの解除の二つの重要な処理を行う必要があります．デバイスの登録解除は，unregister_netdevカーネル・コールでnet_device構造体をカーネルから取り除きます．これにより，上位プロトコル・レイヤからこのデバイスへのインターフェースが断たれることになります．また，割り込みルーチンの解除はfree_irqカーネル・コールで行われます．これにより，割り込みハンドラとIRQの関連を断ち，デバイスからの割り込みを無視するようになります．

2 ネットワーク・ドライバを記述する

IF_NIC向けのEthernetドライバを実際に作ってみましょう．

● デバイスとドライバのプロトコルを決めよう！

まずはIF_NICのドライバを作成するうえで，考慮すべき特徴を考えてみましょう．

- ターゲット動作だけが可能なPCIデバイスである
- 送信・受信はデバイスのバッファを介して行われる
- 受信・送信完了・エラーは割り込みで通知される

以上3点から，どのようなことをやればよいのか，ほぼ想像がついた読者も多いことでしょう．

1) PCIデバイスとして認識し，デバイスのメモリ空間をドライバにマップすることで，ドライバから直接，デバイスのメモリ空間を読み書きできるようにする必要があります．
2) 送信用メソッド［start_xmit()］は，ソケット・バッファを介して渡されてきたデータをIF_NICの送信用バッファに書き込み，送信要求を出すようにする必要があります．
3) 受信は割り込みで通知されるので，割り込みハンドラの中で，受信バッファからデータを受信し，ソケット・バッファを作成して，上位のプロトコル・レイヤに渡すようにする必要があります．
4) エラーなどの例外を割り込みハンドラ内で管理する必要があります．

● ドライバを成すメソッドを実装しよう！

リスト1（pp.92-95）に今回作成したIF_NICドライバを示します．どのように実装したかについて，それぞれのドライバ・メソッドを説明していきます．

（1）init_module

この関数では，二つのことを行わなくてはなりません．PCIバスにあるデバイスを認識し，リソースをドライバにマップすることと，net_device構造体を作成し，カーネルに登録することです．

まず，PCIバス上にあるデバイスを探し，デバイスが見つかると，割り込みを設定します．request_irqカーネル・コールが割り込みハンドラ（ifnic_interrupt）をIRQと関連付けています．これで，対応するIRQが発生するとifnic_interruptが呼ばれるようになります．IF_NIC上のレジスタや送受信バッファ空間へのポインタは，PCIコンフィグレーション空間から獲得した物理アドレス情報をioremapカーネル・コールによってカーネル仮想アドレスにupすることで得られます．

次に，net_device構造体をalloc_etherdevカーネルコールで作成し，そのメンバを初期化します．この際，MACアドレスの登録が必要になるので，デバイス上のシリアルROMから読み出し，登録しています．

そのあと，register_netdevカーネル・コールでnet_device構造体をカーネルに登録し，ネットワーク・デバイス・ドライバとしての前処理が完了します．ここまで到達すると，“eth?”という名前のネットワーク・インターフェースがカーネルに登録され，ifconfigコマンドなどで利用できるようになります．

（2）cleanup_module

unregister_netdevカーネル・コールでnet_device構造体をカーネルから取り外し，ネットワーク・デバイスとしての動作を終わりにします．free_irqカーネル・コールはIRQ

---

column

Linuxドライバのデバッグ方法

Linuxの場合，ドライバのデバッグ方法は確立されていないのが実際です．筆者がもっとも信頼している（というか，選ぶ余地がないのだが…）方法として，

1) ソース・コードをしっかり眺め，頭の中でシミュレーションする！
2) printkでドライバのログを出力！
3) dmesgコマンドでその出力を確認！

という三つの操作を繰り返し，力技で作りあげていく方法です．

しかし，非常にクリティカルなバグにはまると，printkの出力タイミングがOoops!メッセージより後になることがあります．この際には，printkによるメッセージがまったく出力されず，マシンがハングしてしまいます．この状況に直面すると，じっくりとソースを眺めることが完成への最短経路という悲しい現実もあります．

組み込み分野でもLinuxが多く使われ始めた昨今，良い方法が出てくることを期待してやまないのが実際のところです．

と割り込みハンドラの関連を切ります．

(3) **ifnic_open**

このメソッドでは，(a)デバイス上の送受信バッファのポインタを設定し，(b)デバイスの送受信機能・割り込みを開始し，(c)上位プロトコル・レイヤからの送信をスタートさせます．この(c)の操作には，netif_start_queueカーネル・コールを使います．

(4) **ifnic_stop**

割り込みのマスク，デバイスの送受信機能の停止を行い，ifnic_openメソッドで確保された動的なメモリ領域を解放します．

(5) **ifnic_start_xmit**

送信の手順は非常に簡単です．引き数として渡されてきたソケット・バッファのデータをデバイスにコピーし，デバイスへの送信要求を行うだけです．ソケット・バッファはドライバの中で利用されたら，freeしてしまってだいじょうぶです．

これで送信は行われますが，上位のプロトコル・レイヤからは次々と送信要求が発行されてきてしまいます．そこで，これ以上の送信バッファの空きがない場合，netif_stop_queueカーネル・コールで送信要求を止めています．

(6) **ifnic_interrupt**

割り込みハンドラでは，本当にそれに関連するハンドラが呼ばれたのかを判断する必要があります．そこで，割り込みの種類を特定するために三つの仕事を行います．

まず一つ目は，送信完了をチェックすることです．送信が完了すると，それに関連する割り込みを解除します．二つ目は受信の割り込みかどうかを判断し，受信したデータをifnic_recv関数で上位プロトコル・レイヤに渡し，割り込みを解除します．上位レイヤにソケット・バッファを渡す前に，Ethernetタイプのデータであることをeth_type_transカーネル・コールでそのソケット・バッファを設定する必要があります．最後はエラー・カウンタなどをアップデートします．リスト1では，差分を変数に足す形でエラーをカウントしています．

(7) **ifnic_recv**

この関数はソケット・バッファに受信データをコピーし，上位プロトコル・レイヤにそれを渡します．

受信用のソケット・バッファは，ドライバがみずからアロケートしなければなりません．そして，ソケット・バッファのdevとlenメンバをアップデートし，netif_rxカーネル・コールで上位プロトコル・レイヤへ受信データを渡しています．

(8) **net_device_stats**

このメソッドは，ネットワーク・デバイスの送受信量やエラーの統計を返します．随時，値の更新を行っているので，単純にifnic_dev->statのポインタを返すだけで十分です．

(9) **ifnic_tx_timeout**

割り込み途中で送信，または受信が完了してしまい，割り込みを解除してしまった際や，デバイスが送信を完了できないときに，送信のタイムアウトが発生します．このような場合のため，このメソッドでは送受信バッファをifnic_interrupt関数と同様にチェックしています．

3 ネットワークの動作確認

● ネットワーク・デバイス・ドライバのコンパイル

動作を確認する際には，ドライバをコンパイルし，できあがったモジュールをロードして，ネットワーク・インターフェースをupさせます．次の手順でコンパイルします．

(1) ドライバのコンパイル

ドライバをコンパイルする際には，カーネル・ソース・コードに含まれるincludeディレクトリが必要です(Red Hat 9では/usr/include/linuxとカーネル・ソースのものが異なっている)．次のコマンドでコンパイル可能です．

```
gcc -DMODULE -D__KERNEL__ -Wall -Wstrict
-prototypes -O6 -I/usr/src/linux/include
-c if_nic.c
```

● ネットワーク・インターフェースの起動

ネットワーク・インターフェースの起動は，次の手順で行います．

(1) ドライバのロード

ドライバをロードするには，

```
/sbin/insmod if_nic.o
```

で可能です．カーネルのバージョンが異なる際には再コンパイルが必要になります．モジュールのロードができたら，

```
/sbin/ifconfig -a
```

で，追加されたインターフェースの名前を確認します(eth?で登録される)．

(2) ネットワーク・インターフェースをup

```
/sbin/ifconfig eth? IPアドレス up
```

IPアドレスは，適当に192.168.2.1などとしてやるとよいでしょう．ネットマスクでは255.255.255.0に設定されます．ネットマスクの指定から，ルーティングも192.168.2.0に設定されるので，一番簡単に実験できる方法が上記です．

● ネットワークの動作確認

ここまで順調にきたら，とりあえずpingで動作を確認してみることができます．ネットワークで接続したほかのマシンから，自分のマシンに対してpingを打ってみてください．

ネットワークがつながっていることを確認できたら，WWWブラウザなりFTPツールなりで，実際にネットワークを使ってみてください．

● ネットワーク・インターフェースの終了

ドライバを取り外すには，次の手順で行ってください．

(1) ネットワーク・インターフェースをdown

次のようにしてネットワークを停止します．

```
/sbin/ifconfig eth? down
```

(2)ドライバのアンロード

ドライバをアンロードするには，

/sbin/rmmod if_nic

とします．

まとめ

正直なところをいうと，当初こんな手作りのLANカードでまともに通信できるとはとても思えませんでしたが，幾多のバグ(おもにハードウェアのバグが多かったが)を乗り越え，はじめてpingが通ったときは，思わずガッツ・ポーズをしてしまいました．特集のタイトルどおり「作りながら学ぶ」が体験できたと思います．

今回のネットワーク・デバイスは，バス・マスタ転送を行わないPCIターゲット・デバイスであるため，ソケット・バッファとPCIデバイス内の送受信バッファの間のデータ転送をCPUが行う必要があります．このデバイスは，非常に基本的な機能しか実装されていないネットワーク・デバイスなので，ドライバも素直に記述できたと思います．

より性能を重視した100Base-TX対応のネットワーク・デバイスでは，PCIバス・マスタ転送に対応するなど，そのデータ転送の制御方法も複雑になりますが，Linuxのネットワーク・ドライバとしての基本構造は同じです．

参考文献

(1) Alessandro Rubini(著)，Jonathan Corbet(著)，山崎 康宏(翻訳)，山崎 邦子(翻訳)，長原 宏治(翻訳)，長原 陽子(翻訳)；LINUXデバイスドライバ 第2版

やまぎわ・しんいち

リスト1 作成したネットワーク・ドライバ

```
// Linux ethernet driver for IF_NIC devices
// Supported for Linux kernel 2.4.x
// (c) Shinichi Yamagiwa (yama@pdmfc.com)
// GNU Public License に従います.

// マルチプロセッサ環境では動作保障しません(というか必ず不具合が出る)
// ので注意
// 複数のIF_NICデバイスが存在している場合は1枚目のみを使う

#define __NO_VERSION__
                    /* don't define kernel_verion in module.h */

#include <linux/module.h>
#include <linux/version.h>

#define IFNIC_PCI_VENDOR_ID 0x6809
#define IFNIC_PCI_DEVICE_ID 0x8010

#define MODULE_NAME "if_nic"
#define NET_NAME "eth%d"

#include <linux/kernel.h> /* printk() */

#include <linux/pci.h>

//ネットワーク関連のインクルード・ファイル
#include <linux/netdevice.h>
#include <linux/etherdevice.h>

// 制御レジスタのサイズ(1MByte)
#define IF_NIC_CTLREG_SIZE 0x100000

// 送信タイムアウト時間の定義
#define TX_TIMEOUT 10*HZ

// 制御レジスタ空間のオフセットの定義
#define DEVSIG 0x000
#define MACCTL 0x008
#define ROMCTL 0x00C
#define INTSTAT 0x010
#define INTMASK 0x014
#define RXCTL 0x020
#define RXERR 0x024
#define RXDROPCNT 0x030
#define RXCRCERRCNT 0x034
#define RXFRMERRCNT 0x038
#define RXBUFCTL0 0x040
#define RXBUFCTL1 0x044
#define RXBUFCTL2 0x048
#define RXBUFCTL3 0x04C
#define TXCTL 0x060
#define TXSTAT 0x064
#define TXCOLCNT 0x06C
#define TXBUFCTL0 0x070
#define TXBUFCTL1 0x074

#define RXBUF_BASE0 0x10000
#define RXBUF_BASE1 0x10800
#define RXBUF_BASE2 0x11000
#define TXBUF_BASE0 0x18000

// デバイス上の送受信バッファの数
#define TXBUF_NUM 1
#define RXBUF_NUM 4

#define TXBUF_NUM_INC(dev) {¥
                  dev->curr_txbuf = (dev->curr_txbuf +
1)%TXBUF_NUM;¥
}
#define RXBUF_NUM_INC(dev) {¥
                  dev->curr_rxbuf = (dev->curr_rxbuf +
1)%RXBUF_NUM;¥
}

// 受信バッファのサイズ
#define RXBUF_MAX 0x8000
// 送信バッファのサイズ
#define TXBUF_MAX 0x8000

struct IFNIC_pci_info{
  unsigned short vendor_id;
  unsigned short device_id;
  unsigned short status_reg;
  unsigned char revision;
  unsigned int  class_code;
  unsigned char header_type;
  unsigned char latency_timer;
  struct resource mapped_mems[12];
  unsigned int card_busp;
  unsigned short subsys_vendor_id;
  unsigned short subsys_id;
  unsigned int extended_rom_base;
  unsigned int new_functionp;
  unsigned char max_latency;
  unsigned char min_grant;
  unsigned char interrupt_pin;
  unsigned char interrupt_line;
};

// IFNIC用のプライベート構造体
struct IFNIC_Dev{
  unsigned char *ctlreg_pbase;
  unsigned char *ctlreg_vbase;
  int irq;
  struct IFNIC_pci_info pci_info;
  int tx_active;
  int curr_txbuf;
  int curr_rxbuf;
```

リスト1 作成したネットワーク・ドライバ(つづき)

```
  int *tx_size; // 送信が失敗(コリジョンの多発)の時に再送するためのサイズ
  unsigned char **txbuf_base;
  unsigned char **rxbuf_base;
  struct net_device_stats stat;
  struct pci_dev *pci_dev;
};

static struct IFNIC_Dev *ifnic_dev;

// net_device構造体
static struct net_device *ifnic_net_dev;

// プロトタイプ
static int ifnic_open(struct net_device *dev);
static int ifnic_start_xmit(struct sk_buff *skb,
                            struct net_device *dev);
static int ifnic_stop(struct net_device *dev);
static struct net_device_stats *ifnic_get_stats(
                                          struct net_device *dev);
static void ifnic_tx_timeout(struct net_device *dev);
static int  _read_serial_rom(unsigned char *p,
                             int offset, int size);
static struct IFNIC_Dev *ifnic_probe(struct IFNIC_Dev *dev);
static void ifnic_interrupt(int irq, void *dev_instance,
                            struct pt_regs *regs);
static int ifnic_recv(struct net_device *dev, int buf_num);

int init_module(void)
{
  int result,irq_result;
  int i;

  printk(KERN_INFO "############## Loading the IF_NIC
                      driver module ..........##########¥n");

  // PCIバスにある IF_NICを探します.
  if((ifnic_dev = ifnic_probe(ifnic_dev)) <= 0){
    printk(KERN_ERR "IF_NIC : can not find
                              a network interface board!¥n");
    return -EINVAL;
  }

  // IRQを登録します.
  if (ifnic_dev->irq >= 0) {
    irq_result = request_irq(ifnic_dev->irq, ifnic_interrupt,
                 SA_SHIRQ, "if_nic", ifnic_dev->pci_dev);
    printk(KERN_INFO "IF_NIC IRQ = %d¥n", ifnic_dev->irq);

    if (irq_result) {
      printk(KERN_INFO "IF_NIC: can't get assigned irq %i¥n",
         ifnic_dev->irq);
      ifnic_dev->irq = -1;
    }
  }
  // デバイスの設定用レジスタ&送受信バッファのベース・アドレスを獲得
  ifnic_dev->ctlreg_vbase = ioremap(((
                    ifnic_dev->pci_info).mapped_mems[0]).start,
                   IF_NIC_CTLREG_SIZE);
  ifnic_dev->ctlreg_pbase = (void *)((
                    ifnic_dev->pci_info).mapped_mems[0]).start;

  printk(KERN_INFO "Control Register base address::
                                    phy=>0x%p, vert=>0x%p¥n",
     ifnic_dev->ctlreg_pbase, ifnic_dev->ctlreg_vbase);

  // グローバル・ポインタの初期化

  printk(KERN_INFO ">>>> Start IF_NIC Ethernet Driver¥n");

  // Ethernet用の net_device構造体を作る
  ifnic_net_dev = alloc_etherdev(sizeof(struct IFNIC_Dev));
  if (!ifnic_net_dev) {
    printk (KERN_ERR "IF_NIC : no memory for ether device¥n");
    return -1;
  }

  // ネットワーク・デバイス構造体に登録
  ifnic_net_dev->open = &ifnic_open;
  ifnic_net_dev->hard_start_xmit = &ifnic_start_xmit;
  ifnic_net_dev->stop = &ifnic_stop;
  ifnic_net_dev->get_stats = &ifnic_get_stats;
  ifnic_net_dev->watchdog_timeo = TX_TIMEOUT;
  ifnic_net_dev->tx_timeout = &ifnic_tx_timeout;

  ifnic_net_dev->priv = ifnic_dev;

  // MACアドレスの読みだし
  if(_read_serial_rom(ifnic_net_dev->dev_addr, 0x10, 6) == -1){
    printk (KERN_ERR "IF_NIC : Can not read MAC Address!!¥n");
    return -1;
  }

  // MACアドレスの表示
  printk(KERN_INFO "IF_NIC: MAC address >> ");
  for (i = 0; i < 5; i++)
    printk(KERN_INFO "%2.2x:", ifnic_net_dev->dev_addr[i]);
  printk(KERN_INFO "%2.2x.¥n", ifnic_net_dev->dev_addr[i]);

  // MACアドレスの設定
  for (i = 0; i < 6; i++)
    *(unsigned int *)(ifnic_dev->ctlreg_vbase + MACCTL) =
                        (1<<31) | ((i & 0x7) << 16)
                      | (ifnic_net_dev->dev_addr[i] & 0xFF);

  // Ethernetデバイスの登録
  result = register_netdev(ifnic_net_dev);
  if (result)
    return -1;

  ifnic_dev->tx_active = 0;

  printk(KERN_INFO "IF_NIC : Registered EtherDev:
         %s (IRQ %d)¥n", ifnic_net_dev->name, ifnic_dev->irq);

  return 0;
}

void cleanup_module(void)
{
  printk(KERN_INFO "#######  Cleanup IF_NIC¥n");
  unregister_netdev(ifnic_net_dev);
  free_irq(ifnic_dev->irq,ifnic_dev->pci_dev);
}

// PCIデバイスの中から IF_NICを探します.
static struct IFNIC_Dev *ifnic_probe(struct IFNIC_Dev *dev)
{
  struct pci_dev *pci_dev = NULL;
  unsigned int val;

  // まずは PCIバスがあるかどうかですが... あるはずですよね?
  if (! pcibios_present())
    return -NODEV;

  while ((pci_dev = pci_find_device(IFNIC_PCI_VENDOR_ID,
                      IFNIC_PCI_DEVICE_ID, pci_dev)) != NULL){
    // このデバイスのためのプライベート構造体を作ります.
    dev = (struct IFNIC_Dev *)kmalloc(sizeof(struct IFNIC_Dev),
                                                GFP_KERNEL);
    memset(dev, 0, sizeof(struct IFNIC_Dev));

    // PCIコンフィグレーション空間の情報を保存します.
    (dev->pci_info).vendor_id = pci_dev->vendor;
    (dev->pci_info).device_id = pci_dev->device;
    (dev->pci_info).header_type = pci_dev->hdr_type;
    (dev->pci_info).class_code = pci_dev->class;
    (dev->pci_info).new_functionp = pci_dev->devfn;
    (dev->pci_info).subsys_vendor_id
                                 = pci_dev->subsystem_vendor;
    (dev->pci_info).subsys_id = pci_dev->subsystem_device;
    (dev->pci_info).extended_rom_base  = pci_dev->rom_base_reg;
    memcpy((dev->pci_info).mapped_mems,pci_dev->resource,
                 sizeof(struct resource)*DEVICE_COUNT_RESOURCE);
    dev->irq = pci_dev->irq;
    dev->pci_dev = pci_dev;

    return dev;
  }

  return NULL;
}

void ifnic_interrupt(int irq, void *dev_instance,
                                         struct pt_regs *regs)
```

リスト 1　作成したネットワーク・ドライバ(つづき)

```
{
  volatile unsigned int *p, *intstat_reg;
  int i;
  unsigned int int_val, tx_buf_val, err_val;

  intstat_reg = (unsigned int *)
                              (ifnic_dev->ctlreg_vbase + INTSTAT);

  // 送信完了をチェック.
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + TXBUFCTL0);
  tx_buf_val = *p;

  int_val = *intstat_reg;

  if((int_val & ((1<<31) | (1 << 16))) != 0){
    for(i=0;i<TXBUF_NUM;i++){
      // 送信完了・エラーをチェック
      if(((tx_buf_val & (1<<31)) == 0) && ((tx_buf_val &
                   (1<<30)) != 0)){ // 送信エラー. このとき再送処理をする.
    *p = (1<<31) | (ifnic_dev->tx_size[i] & 0x7FF);
    ifnic_dev->stat.tx_packets ++;
    ifnic_dev->stat.tx_bytes += ifnic_dev->tx_size[i];

      }
      else if((tx_buf_val & (1<<31)) == 0){ // 送信完了のとき
    ifnic_dev->stat.tx_packets ++;
    ifnic_dev->stat.tx_bytes += ifnic_dev->tx_size[i];
      }
      p ++;
    }
  }

  // 送信バッファに空きができたかチェックして、送信キューを再開
  if((int_val & (1 << 31)) == 0 && ifnic_dev->tx_active == 0){
    netif_wake_queue(ifnic_net_dev);
    ifnic_dev->tx_active = 1;
  }

  // 受信バッファをチェック.
  // 受信のときは受信バッファにたまった順に処理しなければなりません.
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXBUFCTL0);
  for(i=0;i<RXBUF_NUM;i++){
    int checking_buf_no;
    volatile unsigned int *checking_bufctl;

    checking_buf_no = (ifnic_dev->curr_rxbuf + i) % RXBUF_NUM;
    checking_bufctl = p + checking_buf_no;

    // 受信完了・エラーをチェック
    if((*checking_bufctl & ((1<<31) | (1<<30))) != 0){
      if((*checking_bufctl & (1<<30)) != 0) {
    //RXBUF_NUM_INC(ifnic_dev);
    *intstat_reg = (1 << checking_buf_no);
    *checking_bufctl = (1<<31);
    continue;
      }
      else if(ifnic_recv(ifnic_net_dev, checking_buf_no)
                                                    ==-1) break;
      // 受信割り込み解除
      *intstat_reg = (1 << checking_buf_no);
      *checking_bufctl = (1<<31);
    }
  }

  // 受信オーバフロー・エラーのとき、受信部をリセット
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXERR);
  err_val = *p;
  if((err_val & ((1 << 31) | (1 << 30) | (1 << 29))) != 0){
    *p = err_val & (1 << 31) | (1 << 30);
    if((err_val & (1 << 31)) != 0){ // オーバフロー・エラー
      *p = *p | (1<<31);
      p = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXCTL);
      *p = *p | (1<<31);
      *p = *p & ~(1<<31);
      ifnic_dev->curr_rxbuf = 0;
    }
  }

  // エラーのカウントをアップデートしておきます.
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXDROPCNT);
  ifnic_dev->stat.rx_dropped += (*p & 0xFF);
  *p = 0;

  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXCRCERRCNT);
  ifnic_dev->stat.rx_crc_errors += (*p & 0xFF);
  *p = 0;

  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXFRMERRCNT);
  ifnic_dev->stat.rx_frame_errors += (*p & 0xFF);
  *p = 0;

  *intstat_reg = int_val & ~0x7;

}

// openのための関数. ifconfig upで呼ばれます.
static int ifnic_open(struct net_device *dev){
  volatile unsigned int *p;
  int i;

  // 送信サイズの履歴を取るバッファを確保
  if((ifnic_dev->tx_size = (int *)kmalloc(sizeof(int)*TXBUF_NUM,
                                        GFP_KERNEL)) == NULL){
    printk(KERN_INFO "IF_NIC: can not allocate tx_size array.
                                                          ¥n");
    return -1;
  }
  // 送受信バッファのベース・アドレスを設定
  if((ifnic_dev->txbuf_base = (unsigned char
**)kmalloc(sizeof(unsigned char *)*TXBUF_NUM, GFP_KERNEL)) ==
                                                         NULL){
    printk(KERN_INFO "IF_NIC: can not allocate
                              txbuf_base pointer array.¥n");
    return -1;
  }
  if((ifnic_dev->rxbuf_base = (unsigned char
**)kmalloc(sizeof(unsigned char **)*RXBUF_NUM, GFP_KERNEL)) ==
                                                         NULL){
    printk(KERN_INFO "IF_NIC: can not allocate
                            rxbuf_base pointer array.¥n");
    return -1;
  }
  // 割り込みマスクを解除
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + INTMASK);
  *p =  (1 << 31) | (1 << 16) | (1 << 15) | (1 << 3) | (1 << 2)
      | (1 << 1) | (1 << 0); // (1<<31) | (1 << 16) | (1 << 15)
      | (1 << 2) | (1 << 1) | (1 << 0);

  // 送受信を開始
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + TXCTL);
  *p = *p & ~(1<<31);
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXCTL);
  *p = (*p & ~(1<<31)) | (1<<30) | (1<<29);

  // 上位プロトコル・レイヤからの送信をスタートする
  netif_start_queue(dev);

  ifnic_dev->tx_active = 1;

  ifnic_dev->curr_txbuf = 0;
  ifnic_dev->curr_rxbuf = 0;

  return 0;
}

// hard_start_xmitのための関数. 送信時に呼ばれます.
static int ifnic_start_xmit(struct sk_buff *skb,
                                       struct net_device *dev){
  struct IFNIC_Dev *priv_dev = dev->priv;
  volatile unsigned int *curr_txctl;
  int i;

  // フレームが最小サイズを侵してないか確認する. 小さすぎるとパディングする.
  if(skb->len < ETH_ZLEN){
    skb = skb_pad(skb, ETH_ZLEN-skb->len);
    if(skb == NULL){
      printk(KERN_ERR "IF_NIC: needed more memory for
                                                  padding.¥n");
      return -1;
    }
    skb->len += (ETH_ZLEN-skb->len);
  }

  curr_txctl = (unsigned int *)(priv_dev->ctlreg_vbase +
```

リスト 1 作成したネットワーク・ドライバ(つづき)

```
                                 TXBUFCTL0) + priv_dev->curr_txbuf;

  // ここで、送信できないときは致命的なバグです.
  if((*curr_txctl & (1 << 31)) != 0){
    printk(KERN_INFO "TX buffer FULL! BUG! This NEVER HAPPEN!
                                                           ¥n");
    return 1;
  }

  // 送信バッファに書き込んで、
  for(i = 0; i < skb->len; i++)
    *(priv_dev->txbuf_base[priv_dev->curr_txbuf] + i) =
                                              skb->data[i];

  // 送信を開始して、
  *curr_txctl = (1 << 31) | (skb->len & 0x7FF);
  priv_dev->tx_size[priv_dev->curr_txbuf] = skb->len;

  // ソケット・バッファを解放
  dev_kfree_skb(skb);

  // 1個しか送信バッファがないので、常に送信後は上位の送信をストップする.
  priv_dev->tx_active = 0;
  netif_stop_queue(ifnic_net_dev);

  return 0;
}

// stopのための関数. ifconfig downで呼ばれます.
static int ifnic_stop(struct net_device *dev){
  volatile unsigned int *p;

  // 割り込みをマスク
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + INTMASK);
  *p = 0;

  // 送受信をストップ
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXCTL);
  *p = *p + (1<<31);
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + TXCTL);
  *p = *p + (1<<31);

  kfree(ifnic_dev->tx_size);
  kfree(ifnic_dev->txbuf_base);
  kfree(ifnic_dev->rxbuf_base);

  return 0;
}

// getstatsのための関数. ifconfigなどで呼ばれます.
static struct net_device_stats *ifnic_get_stats(struct
net_device *dev){
  return &(ifnic_dev->stat);
}

// 送信タイム・アウトのための関数.
// 割り込み処理をミスった可能性があるので、送信バッファをなめてみる.
static void ifnic_tx_timeout(struct net_device *dev){
  volatile unsigned int *p;
  int i;

  // 送信完了をチェック.
  p = (unsigned int *)(ifnic_dev->ctlreg_vbase + TXBUFCTL0);
  for(i=0;i<TXBUF_NUM;i++){
    // 送信エラーをチェック
    if(((*p & (1<<31)) == 0) && ((*p & (1<<30)) != 0)){
    // 送信エラー. このとき再送処理をする.
      *p = (1<<31) | (ifnic_dev->tx_size[i] & 0x7FF);
    }
    p ++;
  }
}

// 受信用関数. 割り込みハンドラから呼ばれます.
static int ifnic_recv(struct net_device *dev, int buf_num){
  struct sk_buff *skb;
  volatile unsigned int *bufctl;
  int i;
  int pkt_len;

  skb = dev_alloc_skb(RXBUF_MAX);
  if (skb == NULL){ //領域のとれない時は受信しない.
    printk(KERN_WARNING "IF_NIC: Can not allocate skb!
            I need more Memory!!¥n");
    return -1;
  }

  skb->dev = dev; // このskbがIF_NICで使われることを記録する

  // IPパケット用に2バイトずらし、16ビット・アラインする.
  skb_reserve(skb, 2);

  // パケット・サイズを獲得する.
  bufctl = (unsigned int *)(ifnic_dev->ctlreg_vbase + RXBUFCTL0)
                                                      + buf_num;
  pkt_len = *bufctl & 0x7FF;

  for(i = 0; i < pkt_len; i ++){
    skb->data[i] = ifnic_dev->rxbuf_base[buf_num][i];
  }

  skb_put(skb, pkt_len);

  skb->len = pkt_len;

  // 統計をアップデート
  ifnic_dev->stat.rx_packets++;
  ifnic_dev->stat.rx_bytes += pkt_len;

  // 上位プロトコル・レイヤに受信要求
  skb->protocol = eth_type_trans(skb, dev);
  netif_rx(skb);

  return 0;
}

// シリアルROMを読む関数
// pにoffset番地からのsizeバイトを読み出します. 失敗は-1がかえる
static int  _read_serial_rom(unsigned char *p, int offset,
                                                    int size){
  volatile unsigned int *romctl_reg;

  romctl_reg = (unsigned int *)(ifnic_dev->ctlreg_vbase +
                                                    ROMCTL);

  // アクセス中かチェック
  if((*romctl_reg & (1<<31)) != 0) return -1;

  while(size != 0){
    // アドレスセット & 読み出し開始
    *romctl_reg = (1<<31) | (1<<30) | ((offset & 0xFF) << 16);
    // 読み出し待ち
    while((*romctl_reg & (1<<31)) != 0){}
    //データ読み出し
    *p = *romctl_reg & 0xFF;
    offset ++;
    p++;
    size --;
  }
  return 0;
}
```

第6章



組み込み機器向けITRON TOPPERSと組み合わせて使える

組み込みTCP/IPプロトコル・スタックTINETの詳解

阿部 司

組み込み機器では，RTOSとしてITRONが採用されることが多い．ITRONベースの機器をネットワークに接続するには，ITRON上で動作するTCP/IPプロトコル・スタックが必要になる．

ここでは組み込み機器向けプロトコル・スタックとして，FreeBSDのスタックをTOPPERS上で動くようにしたTINETについて解説する．想定しているハードウェアはH8マイコンにRTL8019ASという非常に一般的な構成なので，応用も利きやすいと思われる．（編集部）

はじめに

ここで紹介するTINET[注1]は，苫小牧高等専門学校情報工学科で研究・開発されたオープン・ソースの組み込みシステム用のTCP/IPプロトコル・スタックです．組み込みシステムには，メモリ容量などの制約があるほか，リアルタイムOS（RTOS）を使用することからリアルタイム性も考慮しなければなりません．TINETは，これらの制約を十分に考慮して設計されています．

本稿では，まず，TINETの特徴と配布ファイルの構成を紹介し，本題であるEthernetデバイス・ドライバの実装を解説します．

1 TINETの特徴

TINETは，FreeBSDのTCP/IPプロトコル・スタックをベースに，組み込みシステムとRTOSの制約を考慮して開発を行いました．理由は，FreeBSDの元になったBSD UNIXのTCP/IPプロトコル・スタックが枯れたソフトウェアであり，事実上，ほかのシステムの見本となっていること，制約の少ないライセンスであることからです．また，デバイス・ドライバもFreeBSDをベースにしていますが，特にRTOSのリアルタイム性の制約を考慮して開発しています．

対応するRTOSは，苫小牧高等専門学校情報工学科もメンバになっているTOPPERSプロジェクトで開発されているJSPカーネルです．TOPPERSプロジェクトとJPSカーネルに関しては，本誌でも連載しています．また，応用プログラムとのインターフェースとなるAPIには，ITRON TCP/IP APIを採用しています．

TINETは，組み込みシステムのメモリ容量の制約に対応するため，単一リンクの終端にある組み込みシステムを想定し，単一のネットワーク・インターフェースのみに限定しており，RAMが32Kバイト，ROMが128Kバイト程度の規模の組み込みシステムを対象としています．実際には，RAMが約10Kバイト，ROMが約45Kバイトで，TOPPERS/JSPカーネルと応用プログラムを含めても，このメモリ制約内に十分収まります．

```
config                 # JSPターゲット依存部
    /h8                # H8プロセッサ依存部
        /akih8_3069f   # 秋月H8/3069Fシステム依存部
tinet                  # TINETのルート・ディレクトリ
    /net               # 汎用ネットワークのソース
    /netinet           # TCP/IP本体のソース
    /netdev            # デバイス・ドライバのルート・ディレクトリ
        /if_ed         # NE2000互換NICドライバのソース
    /netapp            # サンプル応用プログラムのソース
    /cfg               # TINETコンフィギュレータ
    /doc               # ドキュメント類
echos                  # ECHOサーバのサンプル
nserv                  # 各種応用プログラムのサンプル
```

図1 TINETのディレクトリ構成

2 TINETのディレクトリとファイルの構成

TINETは，宮城県産業技術総合センタの好意により同センタ（http://www.mit.pref.miyagi.jp/embedded/consortium/）より配布されています．今回は，TOPPERS/JSPリリース1.4に対応したTINETリリース1.1のデバイス・ドライバを解説します．また，実装ターゲットは，秋月電子通商製のH8/3069F LANボードで，NICはNE2000互換のREALTEK製RTL8019ASです．

図1にTINETのディレクトリ構成を示します．主要なディ

注1：名前の由来は，TomakomaiのTに，インターネットの略称であるINETを組み合わせたもの．

表1 TINET内部パラメータ調整用ファイル

ファイル名	ディレクトリ	定義内容
tinet_config.h	tinet	以下のファイルをインクルードする
tinet_app_config.h	$(UNAME)	応用プログラムの依存定義
tinet_cpu_config.h	config/$(CPU)	プロセッサ依存定義
tinet_sys_config.h	config/$(CPU)/$(SYS)	システム依存定義
tinet_nic_config.h	tinet/netdev/$(NIC)	Ethernetデバイス・ドライバ依存定義

表2 TINETに対するハードウェア依存性定義ファイル

ファイル名	ディレクトリ	定義内容
tinet_defs.h	tinet	以下のファイルをインクルードする
tinet_cpu_defs.h	config/$(CPU)	プロセッサ依存定義
tinet_nic_defs.h	tinet/netdev/$(NIC)	Ethernetデバイス・ドライバ依存定義

リスト1 応用プログラムのMakefile

```
#  ネットワーク・インターフェースの選択,
#  いずれか一つ選択する.
#NET_IF = loop
#NET_IF = ppp
NET_IF = ether

#  イーサネット・デバイス・ドライバの選択
NET_DEV = if_ed

#  トランスポート層の選択
SUPPORT_TCP = true
SUPPORT_UDP = true

#  TINET の Makefile.config のインクルード
include $(SRCDIR)/tinet/Makefile.config
```

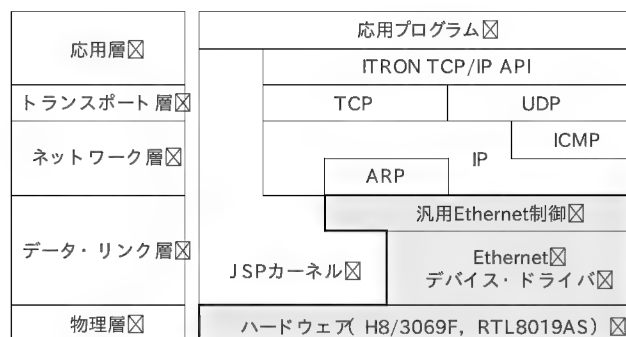


図2 ネットワークの階層構造

レクトリと含まれているファイルの概要を解説しますが，以後，$()で囲われたディレクトリは総称名です．意味と秋月電子通商製H8/3069F LANボードの固有名を以下に示します．

- $(CPU)はプロセッサの総称名：h8
- $(SYS)はシステムの総称名　：akih8_3069f
- $(NIC)はEthernetデバイス・ドライバの総称名：if_ed

ほかに$(UNAME)は応用プログラム名を意味しています．

● 汎用ネットワーク制御に関するファイル

ディレクトリtinet/netには，ネットワーク・システムに依存しない汎用ネットワーク制御に関するファイルが入っています．

● TCP/IPプロトコル・スタック本体のファイル

ディレクトリtinet/netinetには，TCP/IPプロトコル・スタック本体のファイルが入っています．

● Ethernetデバイス・ドライバに関するファイル

Ethernetデバイス・ドライバに関するファイルの中で，JSPターゲットに依存しないファイルは，ディレクトリtinet/netdev/$(NIC)に，JSPターゲットに依存するファイルは，ディレクトリconfig/$(CPU)とconfig/$(CPU)/$(SYS)に入っています．

● TINET内部パラメータ調整用ファイル(表1)

TOPPERS/JSPでは，ターゲット依存部で提供すべきカーネル用のデータ型，関数および定数は，cpu_config.hとsys_config.hにより指定します．TINETにも，これらと同様のファイルがあります．ただし，TOPPERS/JSPとは少し性格が異なり，TINET内部パラメータ調整用ファイルで，ディレクトリtinetにあるtinet_config.hで一括してインクルードしています．表1に，ファイル名，ディレクトリ，および定義内容を示します．詳しい内容は，ディレクトリtinet/docにあるtinet_config.txtを参照してください．

● TINETに対するハードウェア依存性定義ファイル(表2)

TOPPERS/JSPと同様，TINETに対するハードウェア依存性を定義するファイルがあり，ディレクトリtinetにあるtinet_defs.hで一括してインクルードしています．表2に，ファイル名，ディレクトリ，および定義内容を示します．詳しい内容は，ディレクトリtinet/docにあるtinet_defs.txtを参照してください．

● 応用プログラムに関係するファイル

応用プログラムのMakefileには，リスト1に示す各行を追加しなければなりません．ネットワーク・インターフェースがEthernetの場合はNET_IF = etherを選択します．これにより，コンパイル・オプションSUPPORT_ETHERが有効になります．また，Ethernetデバイス・ドライバは，現在if_edのみ選択可能です．

3 ネットワーク階層構造と汎用Ethernet制御

TINETのネットワーク階層を図2に示します．TINETは，

図3　ネットワーク・バッファの構造

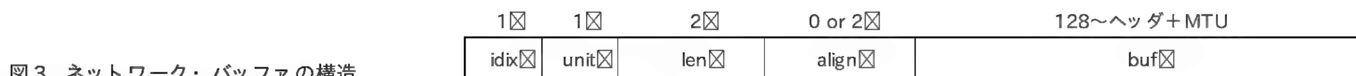


表3　ネットワーク・バッファ構造体のメンバの内容

メンバ名	内　容
idix	固定長メモリ・プールのID番号を保持する配列のインデックス
unit	将来の拡張用，物理ネットワーク層の選択
len	データ長，128，256，512，1,024，およびMTU+ヘッダ
align	IPヘッダ以降を4バイト境界に整列するためのダミー
buf	ネットワーク・フレームのヘッダとペイロード

リスト2　IP出力関数(ip_output)

```
ER ip_output (T_NET_BUF *output, TMO tmout)
{
    /* 途中省略 */
    gw = rtalloc(ntohs(ip4h->dst));
    IF_SET_PROTO(output, IF_PROTO_IP);
    if ((error = IF_OUTPUT(output, (VP)&gw, NULL, tmout)) != E_OK)
        /* 送信エラーを記録する */
        return error;
    }
```

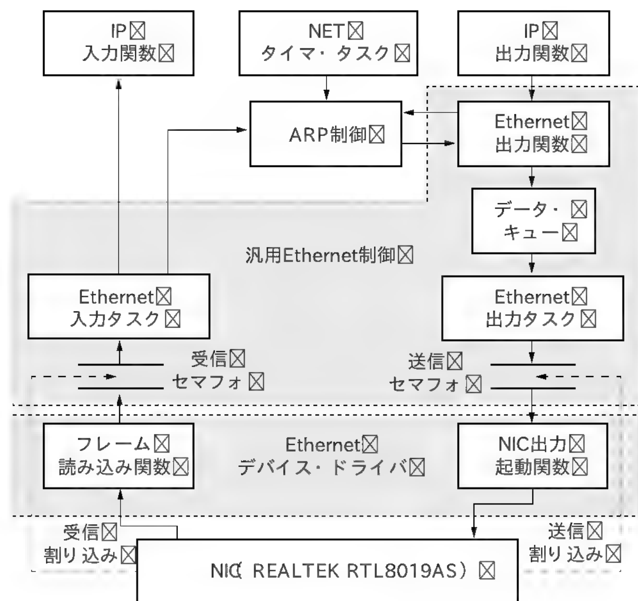


図4　Ethernetの制御とデータの流れ

表4　汎用ネットワーク・インターフェース制御マクロ

マクロ名	Ethernetの定義	設定内容
T_IF_HDR	T_ETHER_HDR	インターフェースのヘッダ構造体の宣言
T_IF_ADDR	T_ETHER_ADDR	インターフェースのアドレスの宣言
IF_MTU	1500	インターフェースのMTU
IF_HDR_ALIGN	2	ヘッダのアライン単位
IF_OUTPUT(o,d,g,t)	ether_output(o,d,g,t)	インターフェースの出力関数(アドレス解決あり)
IF_RAW_OUTPUT(o,t)	ether_raw_output(o,t)	インターフェースの出力関数(アドレス解決なし)
IF_SET_PROTO(b,p)	(GET_ETHER_HDR(b)->type = htons(p))	インターフェースの上位プロトコル設定関数
IF_SOFTC_TO_IFADDR(s)	((T_IF_ADDR*)(s)->ifaddr.lladdr)	ソフトウェア情報からMACアドレスを取り出す関数
IF_IPV4_ADDR_LOCAL	IPV4_ADDR_LOCAL	インターフェースのIPv4アドレス
IF_IPV4_ADDR_LOCAL_BC	IPV4_ADDR_LOCAL_BC	インターフェースのIPv4ブロードキャスト・アドレス
IF_PROTO_IP	ETHER_TYPE_IP	インターフェースのIPプロトコルの値
IF_PROTO_ARP	ETHER_TYPE_ARP	インターフェースのARPプロトコルの値

TOPPERS/JSPに組み込まれるのではなく，TOPPERS/JSPと応用プログラムの中間に位置するミドルウェアです．

● ネットワーク・バッファ

TINETでは，図3に示すように，固定長メモリ・プールから割り当てるネットワーク・バッファを用いて，プロトコル・スタック内における各層間のデータの受け渡しを行います．ネットワーク・バッファ構造体のメンバの内容を表3に示します．送受信いずれも，割り当て前にデータ・サイズを確定し，プロトコル・スタック内では，ネットワーク・バッファの動的メモリ操作は行いません．

● Ethernetの制御とデータの流れ

図4に，Ethernetの制御とデータの流れを示します．図のデータ・リンク層は，NICに依存しない汎用Ethernet制御と，NICに依存するEthernetデバイス・ドライバから構成されています．ここでは，汎用Ethernet制御の動作を，送受信の流れに沿って解説し，Ethernetデバイス・ドライバは，後で詳しく解説します．

● Ethernetへの出力

まず，Ethernetへの出力を見ていきましょう．リスト2に示すIP出力関数ip_outputは，ネットワーク・バッファoutputにEthernetヘッダとIPデータグラムを設定して，ネットワーク・インターフェース出力関数IF_OUTPUTを呼び出します．このIF_OUTPUTは，ネットワーク・インターフェースへの依存を避けるため汎用ネットワーク・インターフェース制

リスト 3　Ethernet出力関数（ether_output）

```
ER ether_output (T_NET_BUF *output, VP dst,
                 T_IF_ADDR *gw, TMO tmout)
{
    /* 送信元 MAC アドレスを設定する */
    ic = IF_ETHER_NIC_GET_SOFTC();
    memcpy(GET_ETHER_HDR(output)->shost,
           ic->ifaddr.lladdr, ETHER_ADDR_LEN);
    switch(ntohs(GET_ETHER_HDR(output)->type)) {
    case ETHER_TYPE_IP:      /* IPv4   */
         if (arp_resolve(&ic->ifaddr, output, *(UW*)dst)) {
              /* TRUE ならアドレス解決済 */
              error = ether_raw_output(output, tmout);
              }
         break;
         }
    return error;
    }
```

御マクロとして定義されており，表4に示すように，ether_output に展開されます．

リスト3に示すEthernet出力関数ether_outputは，ネットワーク・バッファoutputのEthernetヘッダに，送信元のMACアドレスを設定します．次に，Ethernetヘッダに設定されている上位プロトコルにより分岐しますが，上位プロトコルがIPv4のときは，ARPにより送信先のMACアドレスを解決します．解決後，関数ether_raw_outputを呼び出し，ネットワーク・バッファをデータ・キューDTQ_ETHER_OUTPUTに投入します．

リスト4に示すEthernet出力タスクether_output_taskは，データ・キューDTQ_ETHER_OUTPUTからネットワーク・バッファoutputを取り出し，送信セマフォic->semid_txb_readyで，NIC割り込みハンドラif_ed_handlerと同期した後，NIC出力起動関数IF_ETHER_NIC_START（マクロ展開後はed_start）を呼び出します．

● Ethernetからの入力

次に，Ethernetからの入力です．まず，後述するNIC割り込みハンドラif_ed_handlerでは，入力は一切行わず，受信セマフォic->semid_rxb_readyにより，リスト5に示すEthernet入力タスクether_input_taskと同期を取ります．

Ethernet入力タスクether_input_taskは，受信セマフォic->semid_rxb_readyで，NIC割り込みハンドラif_ed_handlerとの同期を行った後，フレーム読み込み関数IF_ETHER_NIC_READ（マクロ展開後はed_read）を呼び出し，受信バッファからEthernetフレームを入力します．次に，Ethernetヘッダに設定されている上位プロトコルによりswitch文で分岐します．

● 汎用ネットワーク・インターフェース制御マクロ

TINETでは，IP層からネットワーク・インターフェースの関数を直接呼び出すことが可能ですが，ネットワーク・インターフェースへの依存を避けるため，各ネットワーク・インターフェースは，表4に示す汎用ネットワーク・インターフェース

リスト 4　Ethernet出力タスク（ether_output_task）

```
void ether_output_task(VP_INT exinf)
{
    ic = IF_ETHER_NIC_GET_SOFTC();
    while (TRUE) {
         while (rcv_dtq(DTQ_ETHER_OUTPUT,
                        (VP_INT*)&output) == E_OK) {
              syscall(wai_sem(ic->semid_txb_ready));
              IF_ETHER_NIC_START(ic, output);
              syscall(rel_net_buf(output));
              }
         }
    }
```

リスト 5　Ethernet入力タスク（ether_input_task）

```
void ether_input_task(VP_INT exinf)
{
    /* トランスポートを初期化する */
    tcp_init();

    /* ネットワーク・インターフェース管理を初期化する */
    ifinit();

    /* ARP を初期化する */
    arp_init();

    /* NIC を初期化する */
    ic = IF_ETHER_NIC_GET_SOFTC();
    IF_ETHER_NIC_PROBE(ic);
    IF_ETHER_NIC_INIT(ic);

    /* Ethernet 出力タスクを起動する */
    syscall(act_tsk(ETHER_OUTPUT_TASK));

    while (TRUE) {
         syscall(wai_sem(ic->semid_rxb_ready));
         if ((input = IF_ETHER_NIC_READ(ic)) != NULL) {
              switch(ntohs(GET_ETHER_HDR(input)->type)) {
              case ETHER_TYPE_IP:      /* IP */
                   ip_input(input);
                   break;
              case ETHER_TYPE_ARP:     /* ARP */
                   arp_input(&ic->ifaddr, input);
                   break;
              default:
                   /* 入力エラーを記録する */
                   syscall(rel_net_buf(input));
                   break;
                   }
              }
         }
    }
```

制御マクロを定義する必要があります．Ethernetの場合は，ディレクトリtinet/netにあるethernet.hに定義されています．

リスト2で示した関数ip_outputでは，ネットワーク・バッファoutputへのEthernetヘッダの設定と，ネットワーク・インターフェース出力関数の呼び出しにマクロが使われており，次のように展開されます．

- IF_SET_PROTOは，送信フレームのヘッダに上位層のプロトコルを設定するマクロで，次のように展開される．

  (GET_ETHER_HDR(output)->type
  =htons(IF_PROTO_IP))

- IF_PROTO_IPは，IPを表すネットワーク・インターフェース固有の上位プロトコル番号に展開されるマクロで，ETHER_TYPE_IPに展開され，さらに0x0800に展開される
- IF_OUTPUTは，ネットワーク・インターフェース出力関数

のマクロで，次のように展開される．

ether_output(output, (VP)&gw, NULL, tmout)

4 NE2000互換NIC REALTEK製RTL8019AS

Ethernetデバイス・ドライバを解説する前に，今回の実装ターゲットとしたRTL8019AS(REALTEK)を解説します．TINETでは，NE2000互換レジスタのみを使用しています．

MACアドレスが書き込まれているシリアルEEPROMと，送受信バッファとして使用されるNIC内蔵SRAMは，レジスタ・ページに関係なく，レジスタ・アドレス0x10でアクセスします．レジスタRSAR0とRSAR1に開始アドレス，レジスタRBCR0とRBCR1に転送バイト数，レジスタCRに読み書き方向を設定し，レジスタ・アドレス0x10を使って読み書きします．シリアルEEPROMのアドレスは0x0000から0x000bです．

RTL8019ASには8ビットモードと16ビットモードがありますが，TINETでは，8ビットモードのみ実装しており，NIC内蔵SRAMのアドレスは0x4000から0x5fffです．なお，デバイス・ドライバからはバイト単位にアクセスしますが，NIC内のフレームの送受信は256バイトを1ページとするページ単位で行われています．

秋月電子通商製のLANボードでは，RTL8019ASのレジスタは0x200000からマップされています．また，割り込みに関しては，RTL8019ASのINT4がH8/3069FのIRQ5に接続されています．

リスト6 TOPPERS/JSPシステム依存アセンブリ定義ファイル(sys_support.S)

```
/* 割り込みベクタへの登録 */
_vectors:
    /* 途中省略 */
    .long     0    /* 16, 0x0040: IRQ4*/
#if defined(SUPPORT_ETHER)
    .long     _if_ed_handler_entry
#else    /* #if defined(SUPPORT_ETHER) */
    .long     0
#endif   /* #if defined(SUPPORT_ETHER) */
                   /* 17, 0x0044: IRQ5*/
    .long     0    /* 18, 0x0048:      */
    /* 途中省略 */

_hardware_init_hook:
    /* 途中省略 */

#if defined(SUPPORT_ETHER)
    /* NIC ハードウェア割り込みの初期設定 */
    mov.l     #H8IER, er0
    mov.b     @er0, r1l
    bset #ED_IER_IP_BIT, r1l
    mov.b     r1l, @er0
#endif   /* #if defined(SUPPORT_ETHER) */
    rts
    /* 途中省略 */

#if defined(SUPPORT_ETHER)
/* NIC ハードウェア割り込み許可 */
_if_ed_handler_enable_int:
_ed_ena_inter:
    mov.b     @ED_IPR, r0l
    bset #ED_IPR_IP_BIT, r0l
    mov.b     r0l, @ED_IPR
    rts

/* NIC ハードウェア割り込み禁止 */
_if_ed_handler_disable_int:
_ed_dis_inter:
    mov.b     @ED_IPR, r0l
    bclr #ED_IPR_IP_BIT, r0l
    mov.b     r0l, @ED_IPR
    rts
#endif   /* #if defined(SUPPORT_ETHER) */
```

5 TOPPERS/JSPターゲットに依存するディレクトリとファイル

NICによって，TOPPERS/JSPのターゲットに依存するファイルは異なると思いますが，TINETのデバイス・ドライバではプロセッサから見たNICのレジスタやメモリのアドレス，割り込みなどをTOPPERS/JSPターゲット依存部のファイルで定義することになります．

NE2000互換NICの場合，TOPPERS/JSPターゲット依存部で関係するファイルは，ディレクトconfig/$(CPU)/$(SYS)にあるTOPPERS/JSPシステム依存アセンブリ関数定義ファイルsys_support.Sと，TOPPERS/JSPシステム依存定義ファイルtinet_sys_config.hです．

● TOPPERS/JSPシステム依存アセンブリ関数定義ファイル(sys_support.S)

リスト6に示すように，割り込み関係の設定を行います．

(1) 割り込みベクタへの登録

LANボードの仕様からRTL8019ASの割り込みラインはH8/3069FのIRQ5に接続されているので，割り込みハンドラを0x0044に登録する．

(2) NICハードウェア割り込みの初期設定

H8の割り込み許可レジスタH8IERのED_IER_IP_BITをセットすることにより割り込み可能に設定する．

(3) NICハードウェア割り込み許可

H8の割り込み優先レジスタED_IPRのED_IPR_IP_BITをセットする．

(4) NICハードウェア割り込み禁止

H8の割り込み優先レジスタED_IPRのED_IPR_IP_BITをクリアする．

リスト7 TOPPERS/JSPシステム依存定義ファイル(tinet_sys_config.h)

```
#define ED_BASE_ADDRESS      0x00200000

#define INHNO_IF_ED          IRQ_EXT5
#define ED_IER_IP_BIT        H8IER_IRQ5E_BIT
#define ED_IPR               H8IPRA
#define ED_IPR_IP_BIT        H8IPR_IRQ5_BIT
```

● TOPPERS/JSPシステム依存定義ファイル（**tinet_sys_config.h**）

リスト7に示すように，NICのレジスタのベース・アドレス，割り込みに関係するレジスタとビットを定義しています．たとえば，sys_support.Sで指定されているED_IPRは，H8/3069FのレジスタH8IPRAに展開されます．

6 TOPPERS/JSPターゲットに依存しないディレクトリとファイル

TINETのデバイス・ドライバでTOPPERS/JSPターゲットに依存しないファイルは，ディレクトtinet/netdev/if_edにあり，NICに依存しないのデバイス・ドライバ共通ファイルと，NICに依存するNIC固有ファイルから構成されています．

表5にデバイス・ドライバ共通ファイルと内容を，表6にNE2000互換NIC固有ファイルと内容を示します．NE2000互換NICの本体プログラムファイルif_ed.cには，これから解説するすべての変数，全域関数および局所関数が定義されています．

表5　デバイス・ドライバ共通ファイル

ファイル名	内　容
Makefile.config	コンパイル・オプション，依存性探索パスおよびオブジェクトの指定
nic.cfg	カーネル・オブジェクトの指定．NE2000互換NICではif_ed.cfgをインクルード
nic_rename.h	デバイス・ドライバ全域名と応用プログラム全域名との衝突回避
tinet_nic_config.h	TINETの内部パラメータ調整用のファイル．nic_rename.hとif_ed.hをインクルード
tinet_nic_defs.h	TINETに対するNICのハードウェア依存性定義ファイル．Ethernetヘッダのアライン調整量の指定

表6　NE2000互換NIC固有ファイル

ファイル名	内　容
if_ed.c	NE2000互換NICの本体プログラム
if_ed.cfg	割り込みハンドラと汎用Ethernet制御との同期用送受信セマフォの定義
if_ed.h	汎用Ethernetデバイス・ドライバ・マクロの定義と関数のプロトタイプ宣言
if_edreg.h	NE2000互換NICのレジスタなどの定義

7 Ethernetデバイス・ドライバ制御構造体と変数

Ethernetデバイス・ドライバを制御するために必要となる変数は，Ethernetデバイス・ドライバ制御構造体で定義しています．この構造体は，NICに依存しないデバイス・ドライバ共通構造体と，NICに依存するNIC固有構造体へのポインタから構成されています．

● デバイス・ドライバ共通構造体（**T_IF_SOFTC**）と変数（**if_soft**）

リスト8に，ディレクトリtinet/netにあるethernet.hで宣言されているデバイス・ドライバ共通構造体T_IF_SOFTCを示します．また，リスト9に，この構造体の実体となる変数if_softを示します．各Ethernetデバイス・ドライバは，T_IF_SOFTCのメンバscのT_IF_ETHER_NIC_SOFTCを定義する必要があります．NE2000互換NICの場合は，表7に示す汎用Ethernetデバイス・ドライバ・マクロにより，struct t_ed_softに展開されますが，依存性を避けるため，実体はif_ed.cに宣言されています．

● NE2000互換NIC固有構造体（**T_ED_SOFTC**）と変数（**ed_soft**）

リスト10に，if_ed.cで宣言されているNE2000互換NIC固有構造体T_ED_SOFTCを示します（表8ではstruct t_ed_softc）．また，リスト11に，この構造体の実体となる

リスト8　デバイス・ドライバ共通構造体宣言（T_IF_SOFTC）

```
struct t_if_softc {
    T_IF_ADDR ifaddr;          /* MAC アドレス              */
    UH   timer;                /* 送信タイマ                */
    T_IF_ETHER_NIC_SOFTC      *sc;
                               /* NIC固有構造体へのポインタ */
    ID   semid_txb_ready;      /* 送信セマフォ              */
    ID   semid_rxb_ready;      /* 受信セマフォ              */
    } T_IF_SOFTC;
```

リスト9　デバイス・ドライバ共通構造体定義（if_soft）

```
T_IF_SOFTC if_softc = {
    {},                        /* MAC アドレス              */
    0,                         /* 送信タイマ                */
    &ed_softc,                 /* NIC固有構造体へのポインタ */
    SEM_IF_ED_SBUF_READY,      /* 送信セマフォ              */
    SEM_IF_ED_RBUF_READY,      /* 受信セマフォ              */
    };
```

表7　汎用Ethernetデバイス・ドライバ・マクロ

マクロ名	展開値	設定内容
IF_ETHER_NIC_START(i,o)	ed_start(i,o)	出力起動関数
IF_ETHER_NIC_GET_SOFTC()	ed_get_softc()	デバイス・ドライバ構造体取得関数
IF_ETHER_NIC_PROBE(i)	ed_probe(i)	NIC検出関数
IF_ETHER_NIC_INIT(i)	ed_init(i)	NIC初期化関数
IF_ETHER_NIC_READ(i)	ed_read(i)	フレーム読み込み関数
IF_ETHER_NIC_RESET(i)	ed_reset(i)	NICリセット関数
IF_ETHER_NIC_WATCHDOG(i)	ed_watchdog(i)	ウォッチドッグ関数
T_IF_ETHER_NIC_SOFTC	struct t_ed_softc	NIC固有構造体宣言

リスト 10　NE2000互換NIC固有構造体宣言（T_ED_SOFTC）

```
typedef struct t_ed_softc {
    UW   nic_addr;      /* NIC のベースアドレス      */
    UW   asic_addr;     /* ASIC のベースアドレス     */
    UH   txb_len[NUM_IF_ED_TXBUF];
                        /* 送信バッファのオクテット数  */
    UB   txb_inuse;     /* 使用中の送信バッファ数     */
    UB   txb_insend;    /* 送信中の送信バッファ数     */
    UB   txb_write;     /* 書き込む送信バッファ       */
    UB   txb_send;      /* 送信する送信バッファ       */
    UB   rxb_read;      /* 読み込む受信ページ         */
    } T_ED_SOFTC;
```

リスト 11　NE2000互換NIC固有構造体定義（ed_softc）

```
static T_ED_SOFTC ed_softc = {
    ED_BASE_ADDRESS + ED_NIC_OFFSET,  /* NIC のベースアドレス   */
    ED_BASE_ADDRESS + ED_ASIC_OFFSET, /* ASIC のベースアドレス  */
    };
```

リスト 12　NE2000互換NICレジスタ定義ファイル（if_edreg.h）

```
#define ED_INT_RAM_SIZE      0x2000
#define ED_INT_RAM_BASE      0x4000

#define ED_PAGE_SIZE         256
#define NUM_IF_ED_TXBUF_PAGE     ¥
    ((ETHER_MAX_LEN + ED_PAGE_SIZE - 1) / ED_PAGE_SIZE)
#define IF_ED_TXBUF_PAGE_SIZE    ¥
    (NUM_IF_ED_TXBUF_PAGE * NUM_IF_ED_TXBUF)
#define IF_ED_RXBUF_PAGE_SIZE    ¥
    (ED_INT_RAM_SIZE / ED_PAGE_SIZE ¥  - IF_ED_TXBUF_PAGE_SIZE)

#define ED_NIC_OFFSET        0x00
#define ED_ASIC_OFFSET       0x10
#define ED_DATA_OFFSET       0x00
```

変数の定義を示します．メンバnic_addrの設定に使われているED_BASE_ADDRESSは，リスト7に示したTOPPERS/JSPシステム依存定義ファイルtinet_sys_config.hに定義されており，値は0x00200000です．ED_NIC_OFFSETは，リスト12に示すNE2000互換NICレジスタ定義ファイルif_edreg.h（送受信バッファとレジスタ・アドレスの定義部分のみ）に定義されており，値は0x00です．したがって，NE2000互換NIC固有構造体t_ed_softcのメンバnic_addrの値は0x00200000となります．同様に，asic_addrの値は0x00200010となり，この値がNICの内蔵SRAMをアクセスするためのアドレスとなります．

図5に，送受信リング・バッファを示します．送受信リング・バッファの設定レジスタには，256バイトを1ページとするページ単位の値を設定します．送受信とも同じNIC内蔵SRAMを使用するので，送信バッファと受信バッファが重ならないようにします．受信バッファの開始ページをレジスタPSTARTに，終了ページ（正確には，終了ページの次のページ）をレジスタPSTOPに設定します．受信フレームは，受信バッファに順次格納され，終了ページに達すると，開始ページに折り返えされるリング・バッファ構造になっています．送信バッファの開始ページは，レジスタTPSRで指定します．図で（ ）内

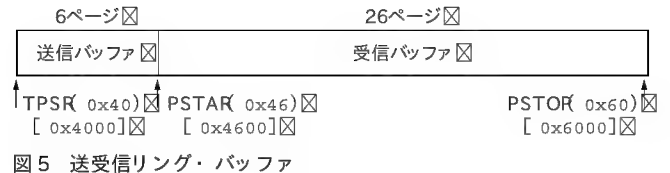


図5　送受信リング・バッファ

表8　NE2000互換NICのEthernetデバイス・ドライバ全域関数

関数名	機　能
if_ed_handler	割り込みハンドラ
ed_start	出力起動関数
ed_get_softc	デバイス・ドライバ共通構造体取得関数
ed_probe	NIC検出関数
ed_init	NIC初期化関数
ed_read	フレーム読み込み関数
ed_reset	NICリセット関数
ed_watchdog	ウォッチドッグ関数

は，ページ・アドレスで，レジスタPSTART，PSTOPおよびTPSRに設定する値です．[ ]内は，デバイス・ドライバから見たバイト単位のアドレスです．

一般に，送信バッファ1個のページ数は，最大長のEthernetフレーム（1518オクテット）が入る6ページ分（1536バイト）を確保します．送信バッファ数NUM_IF_ED_TXBUFはtinet_sys_config.hで1に定義されており，図5に示す送信バッファ全体のページ数IF_ED_TXBUF_PAGE_SIZEは6ページとなります．

また，図5に示す受信バッファ全体のページ数IF_ED_RXBUF_PAGE_SIZEは，NIC内蔵SRAMの残りのページ数で，26ページとなります．送信バッファ数を増加することも可能ですが，そのかわり受信バッファ数が減少することに注意してください．

8 汎用Ethernetデバイス・ドライバ・マクロとNE2000互換NIC全域関数

汎用ネットワーク・インターフェース制御と同様に，汎用Ethernet制御からデバイス・ドライバの各関数を直接呼び出すことは可能ですが，NICへの依存を避けるため，各Ethernetデバイス・ドライバは，表7に示す汎用Ethernetデバイス・ドライバ・マクロを定義する必要があります．NE2000互換NICの場合は，if_ed.hに定義されています．

表8に，NE2000互換NICのEthernetデバイス・ドライバ全域関数を示します．この全域関数は，表の汎用Ethernetデバイス・ドライバ・マクロに含まれており，デバイス・ドライバに必須の関数です．

● NIC割り込みハンドラ(if_ed_handler)

NICからの割り込みは，送信・受信ともにリスト13に示すTOPPERS/JSPカーネルに登録したNIC割り込みハンドラif_ed_handlerで処理します．以下は，動作概要です．

- 送信完了と送信エラーにより送信割り込みが発生します．NIC固有構造体のメンバsc->txb_insendとsc->txb_inuseは，リスト10に示すように，送信中バッファ数と使用中バッファ数を表しており，送信が完了したので，それぞれデクリメントします．次に，sc->txb_inuseが0以上なら，未送信フレームが送信バッファに残っているため，関数ed_xmitで，NICに送信を指示します．最後に，送信セマフォic->semid_txb_readyによりEthernet出力タスクと同期を取ります．
- 受信完了と受信エラーにより受信割り込みが発生します．受信完了は，受信セマフォic->semid_rxb_readyによりEthernet入力タスクと同期を取るだけで終了します．

● NIC出力起動関数(ed_start)

リスト14に，NIC出力起動関数ed_startを示します．ed_start関数ed_pio_writememによりネットワーク・バッファoutputのEthernetフレームを送信バッファに書き込みます．sc->txb_writeは，送信フレームを書き込む送信バッファのインデックスです．

次に，書き込んだオクテット数を，sc->txb_writeをインデックスとして，配列sc->txb_lenに設定します．ここでは，スロット時間[注2]を考慮して，Ethernetフレームが，最短フレーム長である64オクテットからCRCの4オクテットを除いた60オクテットより短いときは，60オクテットを設定します．

送信バッファを切り替えた後，使用中の送信バッファ数であるsc->txb_inuseをインクリメントします．最後に，現在送信中でなければ，関数ed_xmitを呼び出して，Ethernetフレームの送信を開始して終了します．

● デバイス・ドライバ共通構造体取得関数(ed_get_softc)

デバイス・ドライバ共通構造体取得関数は，リスト9に示した局所変数のデバイス・ドライバ共通構造体の実体である変数if_softcを返すだけの関数です．

● NIC検出関数(ed_probe)

リスト15にNIC検出関数を示します．NIC検出関数の本来の機能は，NE2000互換NICがインストールされていることを確かめ，互換NICでもベンダごとに異なるメモリやレジスタの違いを記録することです．しかしTINETでは，コンパイル時にNICが確定しているので，このNIC検出関数では，次に示す操作のみ実行します．

(1) NICをリセットする

注2：Ethernetの制御方式であるCSMA/CDにおける衝突検出のため，送信し続けなければならない最短時間で，64オクテットを送信する時間．

リスト13 NIC割り込みハンドラ(if_ed_handler)

```
void if_ed_handler (void)
{
    /* 送信割り込み処理 */
    if (isr & (ED_ISR_PTX | ED_ISR_TXE)) {
        if (isr & ED_ISR_TXE)
            /* 送信エラーを記録する */

        if (sc->txb_insend)
            sc->txb_insend --;
        if (sc->txb_inuse)
            sc->txb_inuse  --;

        /* 送信タイム・アウトをリセットする。*/
        ic->timer = 0;

        /* まだ送信バッファに残っていれば送信する */
        if (sc->txb_inuse)
            ed_xmit(ic);
        if (isig_sem(ic->semid_txb_ready) != E_OK)
            /* セマフォ・エラーを記録する */
        }

    /* 受信割り込み処理 */
    if (isr & (ED_ISR_PRX | ED_ISR_RXE | ED_ISR_OVW)) {
        if (isr & ED_ISR_OVW)
            /* 上書きエラーとリセットを記録する */
        else {
            if (isr & ED_ISR_RXE)
                /* 受信エラーを記録する */
            if (isig_sem(ic->semid_rxb_ready) != E_OK)
                /* セマフォ・エラーを記録する */
            }
        }
    }
```

リスト14 NIC出力起動関数(ed_start)

```
void ed_start (T_IF_SOFTC *ic, T_NET_BUF *output)
{
    /* 送信バッファに書込む. */
    ed_pio_writemem(sc, output->buf,
                    ED_INT_RAM_BASE
                      + sc->txb_write
                      * NUM_IF_ED_TXBUF_PAGE
                      * ED_PAGE_SIZE,
                    output->len);

    /* 送信バッファに書込んだオクテット数を記録する. */
    if (output->len > ETHER_MIN_LEN - ETHER_CRC_LEN)
        sc->txb_len[sc->txb_write] = output->len;
    else
        sc->txb_len[sc->txb_write] =
            ETHER_MIN_LEN - ETHER_CRC_LEN;

    /* 送信バッファを切り替える. */
    sc->txb_write ++;
    if (sc->txb_write == NUM_IF_ED_TXBUF)
        sc->txb_write = 0;

    sc->txb_inuse ++;

    /* もし送信中でなければ，送信を開始する. */
    if (sc->txb_insend == 0)
        ed_xmit(ic);
    }
```

(2) データ構成レジスタDCRを設定する

(3) シリアルEEPROMからMACアドレスを読み出す

(4) すべての割り込みフラグをリセットする

● NIC初期化関数(ed_init)

NIC初期化関数は，NICからの割り込みを禁止して，初期化本体関数ed_init_subを呼び出します．

リスト 15　NIC 検出関数（ed_probe）

```
void ed_probe (T_IF_SOFTC *ic)
{
    /* リセットする */
    tmp = inb(sc->asic_addr + ED_RESET_OFFSET);
    outb(sc->asic_addr + ED_RESET_OFFSET, tmp);
    dly_tsk(5);

    /* 途中省略 */

    /* DCR レジスタを設定する */
    outb(sc->nic_addr + ED_P0_DCR, ED_DCR_FT1 | ED_DCR_LS);

    /* MAC アドレスを読み込む */
    ed_pio_readmem(sc, 0, romdata, ETHER_ADDR_LEN * 2);
    for (ix = 0; ix < ETHER_ADDR_LEN; ix ++)
        ic->ifaddr.lladdr[ix] = romdata[ix * 2 + 1];

    /* すべての割り込みフラグをリセットする */
    outb(sc->nic_addr + ED_P0_ISR, 0xff);
    }
```

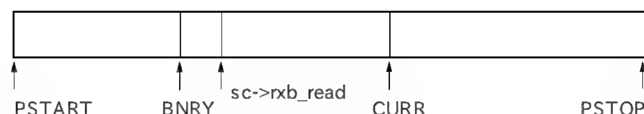


図6　受信バッファ

リスト 16　Ethernet フレーム読み込み関数（ed_read）

```
T_NET_BUF *ed_read (T_IF_SOFTC *ic)
{
   /* 未読のフレームがあるかチェックする. */
   curr = inb(sc->nic_addr + ED_P1_CURR);
   if (sc->rxb_read != curr) {

        /* 受信フレームの先頭を得る. */
        frame_ptr = sc->rxb_read * ED_PAGE_SIZE;

        /* 受信フレーム・ヘッダ構造体を取り出す. */
        ed_pio_readmem(sc, frame_ptr, (char *)&frame_hdr,
                                      sizeof(frame_hdr));

#if SIL_ENDIAN == SIL_ENDIAN_BIG
        frame_hdr.count = (frame_hdr.count << 8)
                        | (frame_hdr.count >> 8);
#endif

        len = frame_hdr.count;
        if (len >  sizeof(T_ED_FRAME_HDR) &&
            len <= IF_MTU + sizeof(T_ETHER_HDR)
                   + sizeof(T_ED_FRAME_HDR) &&
            frame_hdr.next >= ED_INT_RXBUF_START &&
            frame_hdr.next <  ED_INT_RXBUF_STOP)
             input =
                ed_get_frame(sc,
                  frame_ptr + sizeof(T_ED_FRAME_HDR),
                         len - sizeof(T_ED_FRAME_HDR));
        else
             /* NICを停止し、リセット後、割り込みを許可する. */
             return NULL;

        /* 次に読み込むフレームを更新する. */
        sc->rxb_read = frame_hdr.next;

        /* 受信フレームの境界ページを更新する. */
        boundry = sc->rxb_read - 1;
        if (boundry < ED_INT_RXBUF_START)
             boundry = ED_INT_RXBUF_STOP - 1;
        outb(sc->nic_addr + ED_P0_BNRY, boundry);
        }

   /* 受信リング・バッファにデータが残っていれば, 受信処理を継続する. */
   curr = inb(sc->nic_addr + ED_P1_CURR);
   if (sc->rxb_read != curr)
        sig_sem(ic->semid_rxb_ready);
   return input;
   }
```

● **フレーム読み込み関数（ed_read）**

図6に受信バッファを示します．受信バッファの開始ページをレジスタ PSTART に，終了ページをレジスタ PSTOP に設定します．受信フレームは，レジスタ CURR に設定されているページに格納されて行きます．レジスタ CURR は自動的に次のページに進み，終了ページに達すると開始ページに戻ります．レジスタ BNRY は，レジスタ CURR が進むことができる限界ページです．sc->rxb_read は，次にデバイス・ドライバが読み込む受信フレームのページを指しています．

リスト 16に，Ethernet フレーム読み込み関数を示します．以下に動作の概要を示します．

(1) sc->rxb_read とレジスタ CURR の値が異なっていれば，受信フレームの読み込みを開始する
(2) 受信フレームの先頭ページアドレスを計算する
(3) 受信フレームの情報は，リスト 17に示す受信フレーム・ヘッダ構造体 frame_hdr に格納されて，受信フレームの直前に置かれる．この frame_hdr を読み込み，プロセッサのバイト・オーダがビッグ・エンディアンの場合は，受信フレーム長を表す frame_hdr.count のバイトを交換する[注3]
(4) frame_hdr.count と，次の受信フレームのページを表す frame_hdr.next が正しいかチェックし，正しければ受信フレームを読み込む．誤っていれば，NICを停止し，リセットして関数を終了する
(5) sc->rxb_read を frame_hdr.next で更新する
(6) 受信フレームの境界ページを表すレジスタ BNRY に frame_hdr.next の前のページを設定する
(7) もう一度，sc->rxb_read とレジスタ CURR を比較して，値が異なっていれば，まだ未読のフレームがあることを意味しているので，受信セマフォ ic->semid_rxb_ready で，Ethernet 入力タスクと同期を取る

● **NIC リセット関数（ed_reset）**

NIC からの割り込みを禁止して，NIC を停止した後，初期化本体関数 ed_init_sub を呼び出します．

● **ウォッチドッグ関数（ed_watchdog）**

T_IF_SOFTC にある送信タイマ timer には，フレーム送信関数 ed_xmit で，以下のようにタイム・アウトが設定されます．

```
ic->timer = TMO_IF_ED_XMIT;
```

リスト 17　受信リング・バッファ・ヘッダ構造体宣言（T_ED_RING）

```
typedef struct t_ed_ring {
    UB   rsr;      /* 受信ステータス           */
    UB   next;     /* 次の受信フレームのページ   */
    UH   count;    /* 受信フレーム長 (長さ +4)  */
    } T_ED_RING;
```

注3：RTL8019AS のデータシートに記載されているが，レジスタ DCR のビット BOS（Byte Order Select）は未実装である．

表9　NE2000互換NICのEthernetデバイス・ドライバ局所関数

関数名	機　能
ed_pio_readmem	NIC内蔵SRAM読み込み関数
ed_pio_writemem	NIC内蔵SRAM書き込み関数
ed_get_frame	NICからのフレーム読み込み関数
ed_xmit	フレーム送信関数
ed_stop	NIC停止関数
ed_init_sub	NIC初期化本体関数
ed_setrcr	受信構成RCRレジスタ設定関数

このタイマは，ディレクトリtinet/netのif.cにある関数if_slowtimoで監視されており，タイム・アウトすると，この関数が呼び出されます．ウォッチドッグ関数は，NICリセット関数ed_resetを呼び出すだけで終了します．

9 NE2000互換NIC局所関数

表9に，Ethernetデバイス・ドライバ内でのみ使用されるNE2000互換NICのEthernetデバイス・ドライバ局所関数を示します．

● NIC内蔵SRAM読み込み関数（ed_pio_readmem）

リスト18に，NIC内蔵SRAM読み込み関数を示します．まず，レジスタRBCR0とRBCR1に転送数を設定し，レジスタRSAR0とRSAR1に，NIC内蔵SRAMの転送元アドレスを設定します．次に，レジスタCRによりNIC内蔵SRAMの読み込みを選択し，NIC内蔵SRAMをアクセスするレジスタ・アドレス，

sc->asic_addr + ED_DATA_OFFSET

から，転送数分の読み出しを行います．

● NIC内蔵SRAM書き込み関数（ed_pio_writemem）

リスト19に，NIC内蔵SRAM書き込み関数を示します．NIC内蔵SRAM書き込み関数は，読み込みが書き込みに変わるだけで，NIC内蔵SRAM読み込み関数ed_pio_readmemとほぼ同じです．

● NICからのフレーム読み込み関数（ed_get_frame）

Ethernetフレームを読み込むときに，図7に示すように，IPヘッダの先頭を4バイト境界に調整する必要があります．このために，ネットワーク・バッファには，図3と表3に示したように，調整用のメンバalignが宣言されており，表4のIF_HDR_ALIGNの定義にしたがって宣言されます．

Ethernetでは，ヘッダ長が14オクテットであるため，IF_HDR_ALIGNは2と定義されています．また，TCPとUDPの入力では，それぞれの疑似ヘッダとSDU[注4]でチェック・サム

リスト18　NIC内蔵SRAM読み込み関数（ed_pio_readmem）

```
static void
ed_pio_readmem (T_ED_SOFTC *sc, UW src, UB *dst, UH amount)
{
    /* 転送数を設定する. */
    outb(sc->nic_addr + ED_P0_RBCR0, amount);
    outb(sc->nic_addr + ED_P0_RBCR1, amount >> 8);

    /* NIC内蔵SRAMの転送元アドレスを設定する. */
    outb(sc->nic_addr + ED_P0_RSAR0, src);
    outb(sc->nic_addr + ED_P0_RSAR1, src >> 8);

    /* レジスタCRにより読み込みを選択する. */

    /* NIC内蔵SRAMから読み込む */
    while (amount -- > 0)
         *dst ++ = inb(sc->asic_addr + ED_DATA_OFFSET);
    }
```

リスト19　NIC内蔵SRAM書き込み関数（ed_pio_writemem）

```
static void
ed_pio_writemem (T_ED_SOFTC *sc, UB *src, UW dst, UH amount)
{
    /* レジスタISRの完了フラグをリセットする. */

    /* 転送数を設定する. */
    outb(sc->nic_addr + ED_P0_RBCR0, amount);
    outb(sc->nic_addr + ED_P0_RBCR1, amount >> 8);

    /* NIC内蔵SRAMの転送先アドレスを設定する. */
    outb(sc->nic_addr + ED_P0_RSAR0, dst);
    outb(sc->nic_addr + ED_P0_RSAR1, dst >> 8);

    /* レジスタCRにより書込みを選択する. */

    /* NIC内蔵SRAに書き込む */
    while (amount -- > 0)
         outb(sc->asic_addr + ED_DATA_OFFSET, *src ++);

    /* 完了するまで約 240us 待つ. */
    }
```

リスト20　NICからのフレーム読み込み関数（ed_get_frame）

```
static T_NET_BUF *
ed_get_frame (T_ED_SOFTC *sc, UW ring, UH len)
{
    align = ((((len - sizeof(T_IF_HDR)) + 3) >> 2) << 2)
                                        + sizeof(T_IF_HDR);

    error = tget_net_buf(&input, align, TMO_IF_ED_GET_NET_BUF);
    if (error == E_OK && input != NULL) {
         dst = input->buf;
         if (ring + len > ED_INT_RAM_BASE + ED_INT_RAM_SIZE) {
              sublen = (ED_INT_RAM_BASE
                                   + ED_INT_RAM_SIZE) - ring;
              ed_pio_readmem(sc, ring, dst, sublen);
              len -= sublen;
              dst += sublen;
              ring = ED_INT_RXBUF_START * ED_PAGE_SIZE;
              }
         ed_pio_readmem(sc, ring, dst, len);
         }
    else
         /* エラーを記録する */
    return input;
    }
```

1	1	2	2	14	20	20	n
idix	unit	len	align	Ether	IP	TCP	TCP/SDU

4オクテット境界

図7　アライン調整

注4：SDU（Service Data Unit）OSI用語でプロトコルが運ぶデータ（ペイロード）

リスト 21　フレーム送信関数（ed_xmit）

```
static void ed_xmit (T_IF_SOFTC *ic)
{
    /* 送信するページを設定する．*/
    outb(sc->nic_addr + ED_P0_TPSR,ED_INT_TXBUF_START
                     + sc->txb_send * NUM_IF_ED_TXBUF_PAGE);

    /* 送信するオクテット数を設定する．*/
    outb(sc->nic_addr + ED_P0_TBCR0,
         sc->txb_len[sc->txb_send]);
    outb(sc->nic_addr + ED_P0_TBCR1,
         sc->txb_len[sc->txb_send] >> 8);

    /* 送信する．*/
    outb(sc->nic_addr + ED_P0_CR, ED_CR_RD2 |
                       ED_CR_PAGE0 |ED_CR_TXP | ED_CR_STA);

    /* 送信バッファを切り替える．*/
    sc->txb_send ++;
    if (sc->txb_send == NUM_IF_ED_TXBUF)
        sc->txb_send = 0;

    sc->txb_insend ++;

    /* 送信タイム・アウトを設定する．*/
    ic->timer = TMO_IF_ED_XMIT;
    }
```

リスト 23　送受信リング・バッファ割り当ての定義

```
#define ED_INT_TXBUF_START
    (ED_INT_RAM_BASE / ED_PAGE_SIZE)
#define ED_INT_TXBUF_STOP
    (ED_INT_RAM_BASE / ED_PAGE_SIZE
          + IF_ED_TXBUF_PAGE_SIZE)
#define ED_INT_RXBUF_START  ED_INT_TXBUF_STOP
#define ED_INT_RXBUF_STOP
    ((ED_INT_RAM_BASE + ED_INT_RAM_SIZE) / ED_PAGE_SIZE)
```

を計算しますが，図7のSDU長nが4オクテット境界でないときは，SDUの後の4オクテット境界まで0のデータを書き込むので，この分を考慮してネットワーク・バッファを確保します．

リスト20（p.105）にNICからのフレーム読み込み関数を示します．変数alignは，SDUの後の4オクテット境界までのバッファ長です．図6に示したように，受信バッファはリング・バッファのため，受信フレームが終了ページを越えるときは，開始ページに折り返されているので，引き数ringが指すページからレジスタPSTOPが指すページまでと，レジスタPSTARTが指すページから受信フレームの終わりのページまでに分けて，受信バッファから受信フレームを読み込みます．

● フレーム送信関数（ed_xmit）

リスト21にフレーム送信関数を示します．ed_xmitを呼び出す前に，NIC出力起動関数ed_startで，送信フレームを送信バッファに書き込んでいるため，ed_xmitではNICに送信を指示するだけです．以下に動作概要を示します．

（1）送信する送信バッファのページを，レジスタTPSRに設定する．sc->txb_sendは，送信する送信バッファのインデックス

（2）ed_startで設定された配列sc->txb_lenから送信するオクテット数を読み，レジスタTBCR0とTBCR1に設定する

（3）レジスタCRのビットTXPをセットして送信を開始する

リスト 22　NIC初期化本体関数（ed_init_sub）

```
static void ed_init_sub (T_IF_SOFTC *ic)
{
    /* 次に読むフレームのページを設定する．  */
    sc->rxb_read = ED_INT_RXBUF_START + 1;

    /* 送信バッファ制御メンバを設定する．  */
    sc->txb_inuse = sc->txb_insend = 0;
    sc->txb_write = sc->txb_send   = 0;

    /* 送信タイマをリセットする．*/
    ic->timer = 0;

    /* NIC の動作が不安定にならないように送受信と内蔵 SRAM への
       アクセスを停止する．*/
    outb(sc->nic_addr + ED_P0_CR, ED_CR_PAGE0 | ED_CR_STP);
    outb(sc->nic_addr + ED_P0_DCR, ED_DCR_FT1 | ED_DCR_LS);

    outb(sc->nic_addr + ED_P0_RBCR0, 0);
    outb(sc->nic_addr + ED_P0_RBCR1, 0);

    outb(sc->nic_addr + ED_P0_RCR, ED_RCR_MON);
    outb(sc->nic_addr + ED_P0_TCR, ED_TCR_LB0);

    /* 送受信リング・バッファを設定する．  */
    outb(sc->nic_addr + ED_P0_TPSR,    ED_INT_TXBUF_START);
    outb(sc->nic_addr + ED_P0_PSTART, ED_INT_RXBUF_START);
    outb(sc->nic_addr + ED_P0_PSTOP,   ED_INT_RXBUF_STOP);
    outb(sc->nic_addr + ED_P0_BNRY,    ED_INT_RXBUF_START);

    /* 全ての割込みフラグをリセットする．*/
    outb(sc->nic_addr + ED_P0_ISR, 0xff);

    /* 割込み許可を設定する．*/
    outb(sc->nic_addr + ED_P0_IMR, ED_IMR_PRX | ED_IMR_PTX
                | ED_IMR_RXE | ED_IMR_TXE | ED_IMR_OVW);

    /* MAC アドレスを設定する．*/
    for (ix = 0; ix < ETHER_ADDR_LEN; ix ++)
         outb(sc->nic_addr + ED_P1_PAR(ix),
                           ic->ifaddr.lladdr[ix]);

    /* 受信フレームを書き込むページを設定する．*/
    outb(sc->nic_addr + ED_P1_CURR, sc->rxb_read);

    /* 受信構成レジスタ (RCR) を設定する．*/
    ed_setrcr(ic);

    /* ループバック・モードを終了する．*/
    outb(sc->nic_addr + ED_P0_TCR, 0);

    /* 送信可能セマフォを初期化する．*/
    for (ix = NUM_IF_ED_TXBUF; ix --; )
         sig_sem(ic->semid_txb_ready);
    }
```

（4）送信バッファを切り替え，送信中のバッファ数sc->txb_insendをインクリメントする

（5）送信タイム・アウトを設定する

● NIC停止関数（ed_stop）

レジスタCRによりNICを停止し，[5 ms]を限度に，NICがリセット状態から復帰するまで待ちます．

● NIC初期化本体関数（ed_init_sub）

リスト22にNIC初期化本体関数を示します．以下は動作概要です．

（1）レジスタBNRYとの関係で，次に読む受信フレームのページを指すsc->rxb_readは受信バッファの先頭アドレスの次のページに設定する

（2）送信バッファ制御に関係するメンバをゼロ・クリアする

表10　割り込みマスク・レジスタIMR

ビット	割込み要因
PRX	受信(エラーなし)
PTX	送信(エラーなし)
RXE	受信エラー
TXE	コリジョン超過による送信エラー
OVW	受信バッファ超過

表11　受信構成レジスタRCR設定

ビット	受信するフレーム
PRO	すべてのアドレスのフレーム
AM	マルチキャスト・フレーム
AB	ブロードキャスト・フレーム
SEP	エラー・フレーム

(3) 送信タイマをリセットする
(4) 各種レジスタの設定中に動作が不安定にならないように，以下に示す操作によりNICを停止する
- レジスタCRにより送受信を停止
- NIC内蔵SRAMへのアクセスも発生しないように，転送数を表すレジスタRBCRをクリア
- レジスタRCRによりモニタ・モード
- レジスタTCRによりループ・バック・モード

(5) リスト23に示す送受信リング・バッファ割り当てにしたがって，送受信リング・バッファの開始・終了ページ・アドレスを設定する
(6) すべての割り込みフラグをリセットする
(7) 割り込みマスク・レジスタIMRに，表10に示す割り込み許可を設定する
(8) レジスタPAR0からPAR5にMACアドレスを設定する
(9) レジスタCURRに受信フレームを書き込むページを設定する
(10) レジスタRCR設定関数ed_setrcrにより受信構成レジスタRCRを設定する
(11) ループ・バック・モードを終了し，送受信を有効にする
(12) 送信可能セマフォを初期化する

●受信構成レジスタRCR設定関数(ed_setrcr)

リスト24に受信構成レジスタRCR設定関数を示します．内部パラメータ調整用ファイルで指定されているIF_ED_CFG_ACCEPT_ALLにより，設定が異なります．

- 定義されている場合は，マルチキャスト設定レジスタMAR0からMAR7の全ビットを1に設定し，すべてのマルチキャスト・フレームを受信します．また，表11に示すアドレスのフレームを受信するようにレジスタRCRに設定します．
- 定義されていない場合は，MAR0からMAR7の全ビットを0に設定し，すべてのマルチキャスト・フレームを受信しないようにします．また，レジスタRCRはブロードキャスト・フレームのみ受信するように設定します．

リスト24　受信構成レジスタRCR設定関数(ed_setrcr)

```
static void ed_setrcr (T_IF_SOFTC *ic)
{
#ifdef IF_ED_CFG_ACCEPT_ALL

    /* マルチキャストの受信設定 */
    for (ix = 0; ix < 8; ix ++)

        /* マルチキャストをすべて受信する. */
        outb(sc->nic_addr + ED_P1_MAR(ix), 0xff);

    /* マルチキャストとエラー・フレームも受信するように設定する. */
    outb(sc->nic_addr + ED_P0_RCR, ED_RCR_PRO| ED_RCR_AM
                                  | ED_RCR_AB |ED_RCR_SEP);

#else

    /* マルチキャストの受信設定 */
    for (ix = 0; ix < 8; ix ++)

        /* マルチキャストをすべて受信しない. */
        outb(sc->nic_addr + ED_P1_MAR(ix), 0x00);

    /* 自分とブロードキャストのみ受信するように設定する. */
    outb(sc->nic_addr + ED_P0_RCR, ED_RCR_AB);

#endif

    /* NIC を起動する. */
    outb(sc->nic_addr + ED_P0_CR, ED_CR_RD2 | ED_CR_PAGE0
                                             | ED_CR_STA);
    }
```

おわりに

NE2000互換NIC以外のデバイス・ドライバの移植方法や，応用プログラムの作成事例を解説できればよかったのですが，誌面のつごう上，今回はできませんでした．

また，今後のインターネットでは，IPバージョン6(IPv6)への対応が不可欠です．IPv6ではマルチキャストを多用するため，Ethernetデバイス・ドライバもマルチキャスト・フレームへの対応が必須です．現在，IPv6対応TINETの基本部の実装がほぼ終わっており，TINETリリース1.1に続いて，IPv6対応のTINETの配布も開始する予定です．

謝辞　本研究は，次に示す各機関からの支援をいただきました．この場を借りて，各関係機関の皆様に感謝します．
- 財団法人道央産業技術振興機構，テーマ「組込み型制御システム用TCP/IPプロトコルスタックの開発」
- 株式会社NTTドコモ北海道苫小牧支店
- 経済産業省東北経済産業局(委託先管理法人：財団法人みやぎ産業振興機構)，テーマ「組込みシステム・オープンプラットフォームの構築とその実用化開発」

あべ・つかさ　苫小牧工業高等専門学校 情報工学科

第7章



CQ RISC評価キット/PowerPC403を使った

TCP/IPプロトコル・スタックの開発と性能評価

大塚 雄三/並木 美太郎

本章では，10Base-TのEthernetをボード上に標準装備したCQ RISC評価キット/PowerPC403(CqREEK/PPC403)に，組み込み機器向けに最適化したTCP/IPプロトコル・スタックを開発し，その性能評価を行う．残念ながら，CqREEK/PPC403は在庫分を売り切った段階で販売終了となったため現在は入手できないが，プロトコル・スタック自体はC言語で記述され，OSやCPUに依存しない移植性の高いものになっているので，ほかのプラットホームに移植するのも容易だと思われる．　(編集部)

はじめに

近年，プリンタやコピー機といった，LAN接続されてPCなどと通信する機器や，情報家電をネットワークで結び，ホーム・ネットワークを構築するようになってきた．そこで，組み込みシステムの分野においてもTCP/IPによる通信を提供する必要性が高まっている．

ここでは，組み込み用として開発されたRISC CPUのPowerPC403GAを搭載した評価用ボード(CQ RISC評価キットPowerPC403)にTCP/IPプロトコル・スタックを載せてみる．

このPowerPCボードでは，Ethernetコントローラ・チップDP83902(ナショナルセミコンダクター)を搭載しており，NE2000互換として動作する．このボード上に組み込みシステム向けのTCP/IPプロトコル・スタックを載せてみる．

1 プロトコル・スタックの目的

CPUの速度は飛躍的に高速化し，記憶容量も大容量化する傾向にある．PCの世界でも，CPUの速度はGHzを超え，HDDも数百Gバイト単位のものを利用できるようになり，メイン・メモリもGバイトを超えるようになった．それにともなって，PCでは豊富な資源を用いて多くの機能を実現することができるようになった．

一方，組み込みシステムの世界でも，使用できるCPUやメモリ資源は豊富になってきたが，組み込まれる製品のコストを下げるためには，不必要に高性能なプロセッサを使用しないなど，資源をできるだけ節約する努力が必要である．そこで，組み込みシステムは，限られた資源で最大限の性能を発揮することが求められる．

PCの場合，さまざまな用途に用いられることが想定されるので，豊富な機能をもつことが重要である．

しかし，組み込みシステムの場合，用途が限定されているため，不必要な機能をもつことはメモリ消費やオーバ・ヘッドの原因となる．したがって，用途によって不要な機能を削減することによって，メモリの消費を抑え，高速に処理を行うことが求められる．

そこで，上記のようなハードウェア資源に対する制約が厳しいという組み込みシステムの特性を考慮し，より少ないメモリ消費量，CPUパワーの消費で動作するTCP/IPプロトコル・スタックを設計する．

2 BSDソケットの問題点

現在，TCP/IPのプロトコル・スタックとそのAPIとして広く利用されているのがBSDソケットがある．しかしながら，このBSDソケットは組み込みシステムのハードウェア資源について考慮されていないという点において，向いていない．

具体的には，受信用バッファとしてBSDではソケットあたり32Kバイト(default値)のメモリを必要とし，送信時にはアプリケーションとプロトコル・スタック間で最低1回，受信時にはプロトコル・スタックとソケット・バッファ，ソケット・バッファとアプリケーション間の2回のデータ・コピーが行われている．

そのほかにも，TCP/IP以外のプロトコルにも柔軟に対応することができるような設計になっている．

組み込みシステムにおいては，汎用性を高めることより，メモリの消費や処理のオーバ・ヘッドなどを回避するほうが重要である．

また，その実装もハードウェア資源を潤沢に用い，スループットやネットワーク輻輳(ふくそう)を回避するような通信機能を重視したものとなっている．

しかし，組み込みシステムでは，使用できるメモリ量やCPUパワーなど，ハードウェア資源に対する制約が厳しいという点を考慮に入れた設計が必要である．

3 プロトコル・スタックの設計方針

プロトコル・スタックを設計するにあたり，その設計方針を以下に示す．

●TCP/IPによる通信の提供
●組み込みシステム向けに軽量
●むだを省いたAPIの提供

本プロトコル・スタックでは，組み込みシステム向けのTCP/IPによる通信を提供する．

組み込みシステム向けということで，軽量（少容量）であることが求められる．そこで，本プロトコル・スタックでは，通信端点としての利用を前提とし，パケット・フォワーディング機能を省いた．また，プロトコル・スタック内でのデータ・コピーを回避することで，組み込みシステムのハードウェア資源に対する制約が厳しいという点を回避する．

具体的な数値目標として，ターゲットとしてはPowerPCボードで，CPUが33MHz，メモリ使用量は実行時で100Kバイト程度で動作するものを目標とする．また，従来のソケットAPIでは，ゼロ・コピーの枠組みへ収めることができないため，独自のAPIによりサービスを提供するが，その際に，ネットワーク・アプリケーション・プログラマなどのプロトコル・スタックの利用者に負担とならないようなAPIを提供する．

4 本プロトコル・スタックの特徴

本プロトコル・スタックの特徴は，大きく次のとおりである．

●ゼロ・コピー
●機能の限定
●独自のAPI

本プロトコル・スタックの最大の特徴は，プロトコル・スタック内でのデータ・コピーを行わないことである．データ・コピーを回避することで，CPUパワーやメモリの消費量を削減する．

また，本プロトコル・スタックは通信端点での利用を前提としており，パケット・フォワーディング機能は実装しない．通常，通信端点として動作するアプリケーション，たとえばメーラやWebブラウザ，ファイル転送などを利用する際には，パケット・フォワーディング機能を必要としない．とりわけ，組み込みシステム向けということで，限定された用途で利用されることを考慮に入れるとルーティング機能は必須ではないだろう．このルーティングはTCP/IPの処理の中でも大きな割合を占める部分であり，これを避けることでプロトコル・スタック内での処理のオーバ・ヘッドを軽減することができる．

もし，ルータとして利用したい場合には，別途ルータに適した設計を行うべきである．もしくは，本プロトコル・スタックを利用し，アプリケーション層においてルータとしての機能を実装することも可能である．プロトコル・スタック自体にはルータの機能はなくとも，自IPアドレス宛てでないパケットを，あるアプリケーションまで拾い上げ，そのアプリケーションでルーティング・テーブルを参照し，パケットの改変や出力ネットワーク・デバイス・インターフェースの決定などを行うことにより，ルータとして機能することが可能である．

本プロトコル・スタックでは，従来のソケットAPIではコピー・レスの障害となるため，ソケットAPIではなく独自のAPIを採用している．しかし，当然ながらネットワーク・アプリケーション・プログラマなどのプロトコル・スタックの利用者に負担とならないようなAPIを提供する．

● ゼロ・コピー――プロトコル・スタック内でデータ・コピーを行わない

従来のプロトコル・スタックの受信の際の処理手順は，まず受信したパケットをデバイスからプロトコル・スタックへと渡す．プロトコル・スタックでは，そのパケットのヘッダを解析し，行う処理を決定し，多くの場合はさらに上位のプロトコル，またはアプリケーションにデータをコピーする．送信の際は，まず上位アプリケーションなどで作成したデータをプロトコル・スタックのバッファへコピーする．そして，そのコピーされたデータにヘッダなどを付与してパケットとして完成させ，デバイスに渡して送信する．この結果，従来のプロトコル・スタックでは，アプリケーションとの間で少なくとも1回のコピーが起きている．ここで，デバイスがEthernetの場合には，一つのパケットを送信するために，1回のコピーで最大で1460バイトのデータ・コピーが起こることになる．

本プロトコル・スタックでは，このコピーを回避し，メモリ・コピーによるCPU処理や，バッファのためのメモリを節約できるという利点をもつ．

● 独自API――コピー・レスで動作

本プロトコル・スタックでは，独自のAPIを採用している．

これは，従来のソケットAPIでは，UNIXの`open`，`close`，`read`，`write`というファイル入出力の枠組みに収めているが，このAPIではユーザが確保した任意のバッファとデータをコピーするように設計しているため，コピー・レスで動作させることが困難であると判断したためである．

しかし，従来のUNIXの入出力の枠組みには収まらないといっても，ネットワーク・アプリケーション・プログラマなど，プロトコル・スタックの利用者にとって，負担となるようなAPIにするわけではない．むしろ，従来のソケットAPIでは，汎用性を高めるために，通信機能を利用する際，特に利用するプロトコルがTCP/IPだけに限定されるような状況においては，冗長な記述が多いが，本プロトコル・スタックが提供するAPIでは，そのような冗長な記述は必要ではない．

5 プロトコル・スタックの全体構成

本プロトコル・スタックは，図1に示すようにユーザ・イン

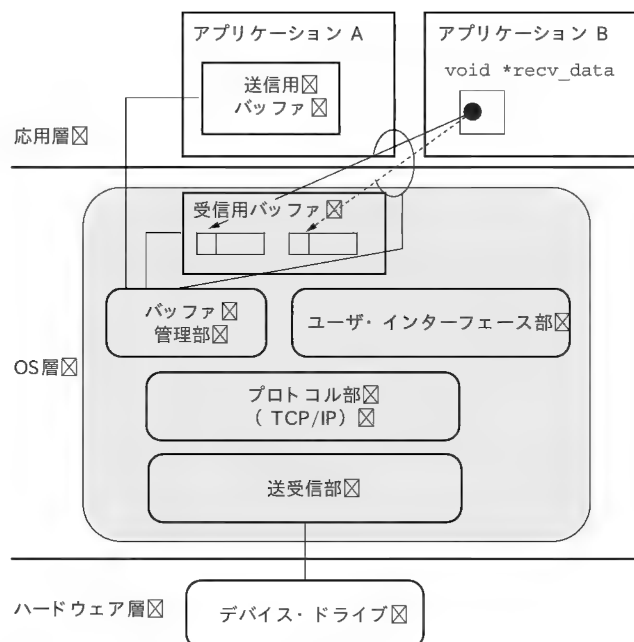


図1 プロトコル・スタックの全体構成図

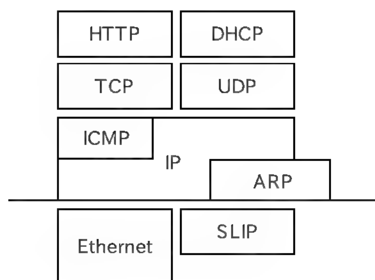


図2
提供しているプロトコル

ターフェース部，バッファ管理部，プロトコル部，送受信部からなっている．

送信用バッファは，それぞれのアプリケーションがもっている．受信用バッファはアプリケーション領域にあるものの，プロトコル・スタックが管理している．その受信用バッファをアプリケーションが直接参照する形になっている．

本プロトコル・スタックでは，TCP/IPによる通信を提供している．提供しているプロトコルは，以下のとおりである．

- HTTP，DHCP
- TCP，UDP
- IP，ICMP，ARP

提供している各層のプロトコルを図2に示した．

サポートしている応用層のレイヤのプロトコルとして，DHCPとHTTPを用意した．

DHCPは，固定のIPアドレスをもたない組み込み機器を，それほど多くの設定を必要とすることなく，ネットワークに接続するだけで利用可能とするためという理由でサポートしている．

HTTPを選択した理由は，一般的に組み込み機器はPCなどと比べ，キーボードやマウス，ディスプレイなどといったインターフェースが乏しい．そのため，PCと同じネットワークに接続して利用するような組み込み機器の設定を行う際には，PCからWebブラウザによりリモートで機器設定を行うことが多々ある．そこで，組み込み用プロトコル・スタックとして，HTTPが必要であると判断した．また，現在エンド・ユーザがPCやインターネットなどにいちばんなじみの深いものとして，Webページの閲覧が挙げられる．

また，本プロトコル・スタックでは，TCP，UDP，IP，ICMP，ARPをサポートしている．

ICMP，ARPのプロトコルは機能が限定されており，ICMPではecho，echo reply，いわゆるpingだけをサポートし，ARPでは下位デバイスとしてEthernetのみをサポートしている．

今回は，ネットワーク・デバイスとしてボードに搭載されているEthernetのみを使用しているが，本プロトコル・スタックは，任意の，かつ同時に複数の下位のネットワーク・デバイスに対応できるものとなっている．

6 プロトコル・スタックの外部仕様

プロトコル・スタックの外部仕様について述べる．

● サポートしているTCP，UDPの機能

本プロトコル・スタックにより，従来同様，送信元，つまりみずからのIPアドレスとポート，そして送信先，つまり相手側のIPアドレスとポートを指定して，TCP，UDPによるデータ送受信を行うことが可能である．

この通信のためのコネクションは，リソースの許す限り複数張ることができる．ただし，送受信の際には一度に送受信するデータ・サイズに注意が必要である．

送信の場合を例に取ると，基本的には，1回の送信コマンド発行ではプロトコル・スタックの判断した送信を行うのに適切なサイズ(＝MSS：Maximum Segment Size)を超える量のデータを送信することができない．これは，送信方法に関わるもので，後述する利用方法の中で詳しく説明するが，プロトコル・スタックを利用するアプリケーション・プログラマは送信したいデータの書き込み位置とサイズは指定されている．その指定されるサイズが，基本的にはMSSを超えることがない．

当然，もっと多量のデータを一度に送信することもできる．だが，上の制限は，本プロトコル・スタックの特徴のコピー・レスに関わるものなので，多少の性能劣化を引き起こす．

受信の際にも，同様のことが当てはまる．

● プロトコル・スタックが用意するAPI

本プロトコル・スタックでは，表1に示した独自のAPIを提供している．これは，従来のソケットAPIでは，ユーザが確保した任意のデータ領域とプロトコル・スタックのバッファとの間でデータをコピーするように設計されているため，ゼロ・コピーで動作させることが困難であると判断したためである．

表1 プロトコル・スタックが用意するAPI一覧

API(C言語仕様)	説　明
typeSock *cre_sock(u_long saddr, int sport, u_long daddr, int dport, char *protocol, int opt);	ディスクリプタを生成する
void del_sock(typeSock *sk);	ディスクリプタを削除する
int ask_rbuf(typeSock *sk, void **p);	受信したパケットのデータのアドレスとサイズを得る．pにアドレスを入れて返される
void read_done(void *p, int len);	データの読み出し終了をプロトコル・スタックに通知する
int set_sbuf(typeSock *sk, void *buf_addr, int buf_len);	送信用バッファを登録する
int ask_sbuf(typeSock *sk, void **p);	送信したいデータの書き込みアドレスとサイズを得る．pにアドレスを入れて返される
int send(typeSock *sk, void *p, int len, int opt);	送信する

また，その際に従来のソケットAPIの冗長な手順を省いた．ソケットAPIでは，前準備としてsocket()，bind()，listen()，accept()や，その引き数にAF_INET，また確保した構造体sockaddr_inの初期化などの手順が必要である．

TCP/IPでの通信を行う場合には，これらの多くは，ほとんどがおまじない程度の冗長なものとなっている．本プロトコル・スタックのcre_sock()は，従来のソケットAPIのsocket()，bind()，listen()，accept()，connect()を含んでいる．前準備として必要なことは，cre_sock()によるディスクリプタの生成と，送信の場合にはバッファの確保と，その登録であるset_sbuf()だけである．

本プロトコル・スタックでは，データの読み書きの前には，どこを読み書きするべきかをAPIのask_rbuf()，ask_sbuf()により得る必要がある．受信データを読み出す場合は，当然ながら送信したいデータを書き込む場合にも，そのアドレスが限定される．これはコピー回避とバッファのデータ構造に関連しており，データ読み書きの際に多少の制限が付くものの，これにより本プロトコル・スタックではコピーを回避することが可能になる．

本プロトコル・スタックには，従来のソケットに比べ大きく異なるread_done()というAPIが用意されている．従来のプロトコル・スタックでは，パケットが到着すると，受信用領域に格納され，解析が行われる．その結果，宛て先のコネクションが決定すると，そのソケット・バッファにデータがコピーされ，受信したパケットは受信用領域からは解放される．

しかし，本プロトコル・スタックでは解析が行われ，宛て先のコネクションが決定した後でも，パケットを受信用領域に保持しておく．そして，アプリケーションからこの領域に保持してあるパケットのデータを直接参照させる．こうすることでコピーを回避しているものの，このパケット・データ領域をいつ解放してよいのかを，プロトコル・スタックからは判断することができない．そこで，アプリケーションが不要となったことを明示的に通知する必要がある．そのためのAPIがread_done()である．

次に，簡単なHTTPクライアント・プログラムを例に，APIの利用方法を説明する．

リスト1 HTTPクライアント・プログラム

```
int test_http_client(u_long daddr, int dport){
    typeSock *sk;
    char *method = "GET";
    char *path = "/";
    char sbuf[128];
    int  empty_size;
    char *send_data;
    char *recv_data;
    int  recv_len;
    char *temp;

    sk = cre_sock(0, 0, daddr, dport, "TCP", 0);   ←Ⓐ

    if(sk==NULL){
        printf("failed at cre_sock()¥n");
        return -1;
    }

    set_sbuf(sk, sbuf, sizeof(sbuf)); ←――――――Ⓑ

    empty_size = ask_sbuf(sk, &send_data);

    if(empty_size < 20){
        printf("not enough sbuf empty¥n");
        return -1;
    }

    sprintf(send_data, "%s %s HTTP/1.0¥r¥n¥r¥n",
                                method, path);        Ⓒ

    send(sk, send_data, sizeof(send_data));

    recv_len = ask_rbuf(sk, &recv_data);

    if(recv_len < 0){
        printf("failed at recv. recv_len<0.¥n");
        return -1;
    }

    temp = recv_data;
    for(i=0; i<recv_len; i++){
        putchar(*temp++);
    }

    read_done(recv_data, recv_len);

    del_sock(sk);

    printf("end: test_http_client¥n");
}
```

リスト1に示すテスト・プログラムは，接続先のIPとポート(普通は80)を引き数とし，そのIP，ポートに対してGETコマンドを発行する．そして，そのレスポンスを表示するというものである．

リスト1のⒶで，APIのcre_sock()により，接続先のIPとポートを指定して，TCPのコネクションをオープンする．次に，Ⓑではあらかじめchar sbuf[128];と送信用バッファとして確保しておいた領域をAPIのset_sbuf()を用いて，プロトコル・スタックへと登録している．ここまでで，送信のための準備が整った．

送信する際には，まず送信したいデータを書き込むべきアドレスとサイズを得るため，APIのask_sbuf()を発行している．このAPIにより得た，send_data，empty_sizeを元にして，sprintf()により送信したいデータを書き込み，APIのsend()で送信している．以上で，HTTPのGETコマンドが相手先に送信される．

次に，HTTPのレスポンスを受信する．受信の際にも，読み出すべきアドレスとサイズを得るため，APIのask_rbuf()を発行する．このAPIにより得たrecv_data，recv_lenを元に，受信データを読み出す．このテスト・プログラムでは，putchar()により，文字として出力しているだけである．

アプリケーション・プログラムが受信データの読み出しを完了したら，その旨をAPIのread_done()により，プロトコル・スタックへ通知しなくてはならない．

その後，del_sock()により，不要となったディスクリプタを削除する．

● プロトコル・スタックの利用方法

前節では，APIの使用例をHTTPクライアント・プログラムを例にして説明したが，ここでは本プロトコル・スタックを利用した送受信の手順について，もう少し深く説明する．

▶ 送信の手順

本プロトコル・スタックを利用してTCP，UDPによる送信を行う際に，従来と異なる点として，まず送信用バッファをアプリケーションがみずから確保しなくてはならないという点がある．

従来では，送信したいデータを用意したら，そのデータ領域へのポインタとサイズ，そしてソケット・ディスクリプタを指定して送信コマンドを発行することで，送信するというスタイルであり，その送信処理の際にプロトコル・スタックのもつ送信用バッファへとデータがコピーされる．そのプロトコル・スタックで送信処理の際に利用する送信用バッファの確保は，当然，プロトコル・スタックの行うことであり，アプリケーション・プログラマの行うことではなかった．アプリケーション・プログラマがプロトコル・スタックのためにバッファを確保する必要はなく，送信したいデータも任意の場所に用意することができた．

しかし，本プロトコル・スタックでは，データをコピーしないということが特徴である．そのため，アプリケーションで送信したいデータのためのデータ領域と，プロトコル・スタックで送信のために利用する送信用バッファを別に用意することはしない．データ・コピーを行わないので，この二つは統合されるべきである．そこで，本プロトコル・スタックでは，プロトコル・スタックで送信の際に利用する送信用バッファは，データの送信を要求するアプリケーションに確保してもらい，それをアプリケーションとプロトコル・スタックで利用することにした．

そのため，データの送信を要求するアプリケーションから見た送信の手順も従来とは異なり，次の手順で行われる．

1) 送信用バッファの確保
2) ソケット・ディスクリプタを得る
3) 送信用バッファをプロトコル・スタックへ登録する
4) 送信データの書き込み位置を得る
5) 送信データ書き込み
6) 送信コマンド発行

まず，先述のとおり，送信の際に利用される送信用バッファは，送信を要求するアプリケーションが確保する．次に，従来と同様，送信のためのディスクリプタを得る．そして，確保しておいた送信用バッファをプロトコル・スタックへ登録する．この送信用バッファをアプリケーションとプロトコル・スタックで利用するとはいえ，管理はプロトコル・スタックが行うべきである．そのため，確保した送信バッファをプロトコル・スタックに登録するという手順が必要となる．

次に，アプリケーションが送信したいデータを送信用バッファに書き込むのだが，送信用バッファの任意の位置に送信したいデータを書き込んではならない．送信したいデータの書き込む位置とサイズはプロトコル・スタックによって指定される．この位置とサイズは後述のAPIにより得ることができる．

アプリケーションでは，APIにより得た位置とサイズを元に送信したいデータを書き込む．そして，その送信したいデータを書き込んだ位置とサイズを引き数とし，送信コマンドを発行する．

以上が送信の手順となる．

▶ 受信の手順

次に，本プロトコル・スタックを利用してTCP，UDPによる受信を行う際の手順について述べる．

従来のソケットAPIを利用した受信の手順は，アプリケーション・プログラムで用意した受信データを収めるためのデータ領域へのポインタとサイズ，そしてソケット・ディスクリプタを指定して，read()などの受信コマンドを発行することで，用意したデータ領域へパケットのデータがコピーされるといったものである．

本プロトコル・スタックでは，送信の際と同様，受信の際にもデータ・コピーを行わないので，従来の方法ではうまくいかない．

そこで，本プロトコル・スタックでは，次のような手順で受信を行う．

1) 受信データの読み出し位置を得る
2) 受信データ読み出し

3) 読み出し完了を通知

本プロトコル・スタックではデータ・コピーを避けるため，受信データの読み出しは，プロトコル・スタックが管理し，受信パケットを収める受信用バッファをアプリケーション・プログラムから直接参照する．その際，アプリケーション・プログラムは後述するAPIを利用し，受信データの位置とサイズを得る．アプリケーション・プログラムは，その位置とサイズから受信データを読み出し，利用する．

そして，受信データの読み出しが終了したら，その旨をプロトコル・スタックへAPIを利用して通知する．これは，従来のものにはない手順である．本プロトコル・スタックではアプリケーション・プログラムが読み出すまで，受信データは受信用バッファに保持されており，アプリケーション・プログラムからこの領域に保持してあるパケットのデータを直接参照させる．

しかし，このパケット・データ領域をいつ解放してよいのかを，プロトコル・スタックからは判断することができない．そこで，アプリケーション・プログラムから不要となったことを明示的に通知する必要がある．

7 プロトコル・スタックの内部仕様

● プロトコル・スタックにおける受信処理

ここでは，本プロトコル・スタックでの受信の際の内部処理について説明する．受信は次の手順で処理される．

1) パケット受信

パケットが受信されると，デバイスを通してプロトコル・スタック中の受信バッファへパケットが格納される．

2) ヘッダの解析

プロトコル・スタックではヘッダを解析し，それにしたがって処理を進める．多くの場合は，さらに上のプロトコル，アプリケーションにデータに渡すことになる．

3) 受信データの位置情報を要求〔API： ask_rbuf()〕

受信データがある場合は次へ．

4) データへのアドレスとサイズを示す

データを渡す方法として，本プロトコル・スタックではデータをコピーするのではなく，受信したパケットをそのままプロトコル・スタック内の受信バッファに留めておき，アプリケーションにはそのプロトコル・スタック中のバッファにあるデータの先頭アドレスを渡す．

5) データの読み出し

アプリケーションは，プロトコル・スタック中のデータを直接参照する．

6) 読み出し終了を通知〔API： read_done()〕

アプリケーションから読み終わったことを示すread_done()を受ける．

7) 領域解放

パケット・データ領域を解放する．

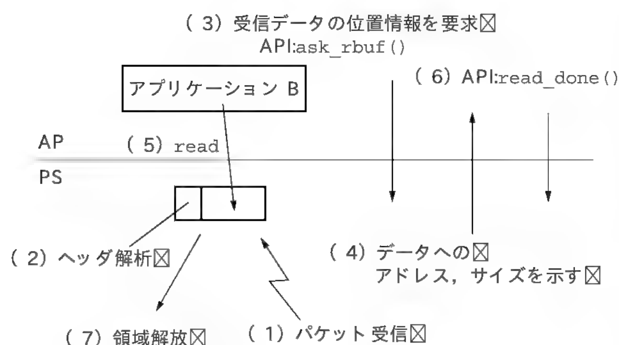


図3 パケット受信処理の図

以上の挙動を図3に示す．

プロトコル・スタック内のバッファ，プロトコル・スタックが管理し，システムのすべての受信パケットが格納されるメモリ領域に対して，通常のユーザ・プログラムから直接アクセスするということは，保護の面で問題となる．しかし，組み込みシステム向けということで，保護よりも性能を重視することにした．

この方法により，従来では受信の際にEthernetの場合で最大1460バイトのメモリ・コピーが2回起きていたが，本プロトコル・スタックでは回避される．

● プロトコル・スタックにおける送信処理

本節では，本プロトコル・スタックでの送信の際の内部処理について説明する．送信は次の手順で行われる．

1) バッファを確保し，登録する〔API： set_sbuf()〕

送信のためのバッファ領域は，アプリケーションで確保し，それをプロトコル・スタックに登録する．以降，この送信用のバッファはプロトコル・スタックで管理するので，その中がどうなっているかはアプリケーション・プログラマにとってはブラック・ボックスとなっている．

2) 送信データの書き込み位置情報を要求〔API： ask_sbuf()〕

アプリケーションからデータを送信する際には，まず送信用のバッファのどこへ送信データを書き込むべきかを，ask_sbuf()を用いてプロトコル・スタックにたずねる．バッファに空きがあれば次へ．

3) 空き領域からヘッダの分を取り分ける

プロトコル・スタックでは，バッファの空き領域からヘッダのための領域を取り分けておき，パケットのデータ部分の先頭にあたるアドレスを示す．

4) データを書き込む

プロトコル・スタックから示された書き込む先頭アドレス，空き領域のサイズに従って，データを書き込む．

5) 送信要求〔API： send()〕

データの書き込みが終わったら，先頭アドレスと書き込んだサイズをsend()を用いてプロトコル・スタックに通知し，送信してもらう．

6) ヘッダの付与などの処理を行い，送信する

プロトコル・スタックでは，そのデータにヘッダを付与し，パケットとして完成させ，デバイスへ渡して送信する．

以上の挙動を図4に示す．

この方法により，送信の場合も受信の場合と同様，データ・コピーが起こらないので，従来ではEthernetの場合で最大1460バイトのメモリ・コピーが回避される．

8 プロトコル・スタックの実装

本プロトコル・スタックは，C言語で作成した．本プロトコル・スタックは，およそ2000行ほどである．

本プロトコル・スタックは，OSは筆者の研究室で開発されている独自のOS「開聞」上での動作を前提としている．しかし，OSの依存性を低くしてあるので，ほかのOSでも動作可能と思われる．

● PowerPCボードの概要

図5にPowerPCボードのブロック図を，表2に仕様を示す．

● PowerPCボードにおけるDP83902の利用の際の注意点

開発用PCからのユーザ・プログラムのダウンロードにEthernetデバイスを利用している関係か，筆者が試したかぎりでは，元のバージョンのBIOSではEthernetデバイスをBIOSを介さずにユーザ・プログラムから利用することができなかった．そこで，まずこのボード上でEthernetデバイスを利用する前に，過去の本誌の記事(1998年5月号，p.204～)やWebページなどを参考に，BIOSとターミナル制御ソフトのバージョン・アップを行ってほしい．このバージョン・アップにより，次の機能が追加される．

1) ユーザ・プログラムのフラッシュ・メモリ上への保管
2) セット時のユーザ・プログラムからの自動起動
3) 起動デバイスの優先順位の設定
4) ユーザ・プログラム名のディレクトリ指定
5) モニタ・プログラムのメニュー改良

ユーザ・プログラムを起動する前にモニタ・プログラムからEthernetデバイスをディセーブルに設定することで，ユーザ・プログラムからEthernetデバイスをBIOSを介さずに利用したいのだが，旧バージョンでは，ユーザ・プログラムを開発用PCからボード上のメモリにダウンロードと，ダウンロードしたユーザ・プログラムの実行が一体化していた．当然，ユーザ・プログラムのダウンロード前にEthernetデバイスをディセーブルにしてしまうと，ユーザ・プログラムのダウンロードが行えなくなる．

そこで，このバージョン・アップを行うことにより，ユーザ・プログラムをダウンロードし，フラッシュ・メモリ上に保管するという手順と，そのフラッシュ・メモリ上のユーザ・プログラムを起動させるという手順に分けられる．そのため，ユーザ・プログラムのダウンロード後にEthernetデバイスをディセーブルに設定し，その後フラッシュ・メモリからユーザ・プログラムを起動することが可能となる．

後は，ユーザ・プログラム中でDP83902の初期化などを行うことで，ユーザ・プログラムから自由にEthernetデバイスを利用することが可能となる．

9 プロトコル・スタックの性能評価

今までに説明した設計に基づいて実装し，その評価を行った．

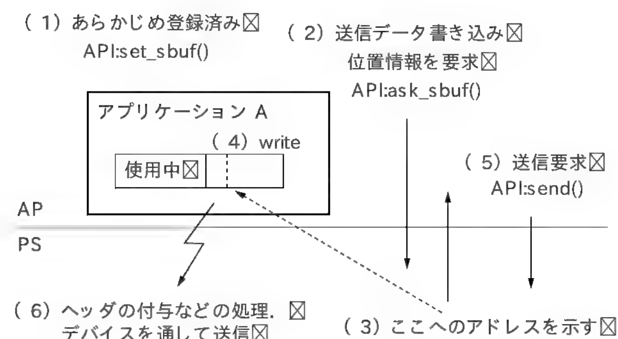


図4　パケット送信処理の図

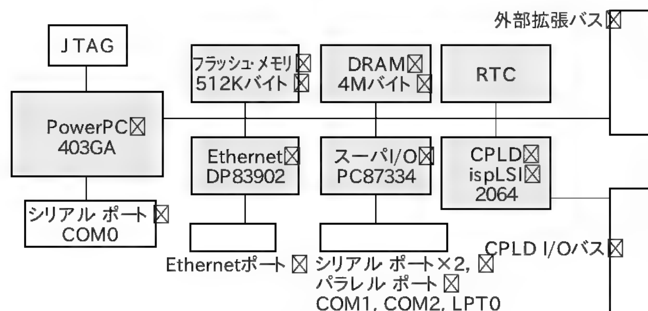


図5　PowerPCボードのブロック図

表2　PowerPCボードの仕様

機能ブロック	仕　様	備　考
CPU	PowerPC 403GA	クロック周波数33.333MHz
メモリ(ROM)	フラッシュ・メモリ 512Kバイト	BIOS ROM用256Kバイト / ユーザ用256Kバイト
メモリ(RAM)	DRAM　4Mバイト	アクセス速度70ns, さらに4Mまで増設可能
Ethernet	10Base-T	NE2000互換
シリアル・ポート	16550互換	3ポート
パラレル・ポート	セントロニクス互換	1ポート
リアルタイム・クロック	RTC-4543	年月日，時分秒
IDE HDDポート	あり	BIOS ROMでは未対応
CPLD　I/Oポート	プログラマブルI/O 15本	ispLSI2064搭載

表3
送信時の時間計測結果

SEND SIZE (バイト)	LOOP CONT (回数)	時　間 (ms)	メモリ・コピーにかかる時間(計算値)(ms)	コピー・レスによる送信処理時間(計算値)
100	1000	453	60	0.88＝453/(453＋60)
500	1000	521	300	0.63＝521/(521＋300)
1460	1000	614	880	0.41＝614/(614＋880)

ここではその結果を示す．

本プロトコル・スタックのコード・サイズは，約80Kバイトとなった．これは，簡易OS機能をもったカーネルを含めたものである．また，実行時のデータ・サイズはコンフィグによって異なるが，最低で10Kバイト弱であり，今回の評価実験の際のデータ・サイズは約20Kバイトだった．

● コピー・レスによる性能向上の評価

ここでは，データ・コピーが行われないことが効いてくる部分の測定を行う．まず，実装環境上でDRAMメモリ・コピーにかかる時間を測定してみたところ，1518バイトをコピーするまでに0.9msかかった．

次に，プロトコル・スタックでデータのコピーが起きていた個所について考える．従来のソケットで送信時にコピーが生じていた個所は`write()`などの送信コマンドを発行した際である．その処理の中で，アプリケーションから示されたアドレスのバッファからデータをソケットのバッファへコピーしていた．

本プロトコル・スタックでの送信コマンドの発行はAPIの`send()`にあたる．そこで，本プロトコル・スタックでパケットの送信にかかる時間を以下の方法で計測する．以下の方法では，結果値を妥当なものとするため1000回送信した時間を計測している．

本システムでは従来のものと異なり，送信の前に書き込むべきアドレスを得る必要がある．そこで，送信時の時間計測部分を`send()`だけでなく，`ask_sbuf()`も含めて計測した．

前出のHTTPクライアント・プログラムの例で示すと，リスト1の©の部分全体を時間計測に用いた(表3)．

この結果よりコピーを行っていないが，処理時間と送信するサイズには若干の相関があることがわかる．本システムでは，コピーを行わないため，大きなサイズを送信する場合には，非常に有効であることが確認できた．特に，今回実装したボードに載っているEthernetで送れる最大サイズ1460バイトを1回送信する場合の処理時間(614μs)に対して，もしコピーを行うならばその処理に880μs余分にかかることになり，全体で1494μsかかることになる．コピーを回避したことで，本来1494μsかかるところが614μsに，つまり送信の際の処理にかかる時間を約4割ほどに短縮できた．

また，時間だけでなく，メモリ・コピーにかかるCPUパワーやメモリの使用量といったハードウェア資源も節約されている．

まとめ

組み込み用として開発されたRISC CPUのPowerPC403GAを搭載した評価用ボードCQ RISC評価キットPowerPC403にTCP/IPプロトコル・スタックを載せてみた．

TCP/IPプロトコル・スタックは，データ・コピーを回避したものとし，また実装する機能を限定したことで，処理軽減を行った．

おおつか・ゆうぞう/なみき・みたろう　東京農工大学大学院 工学研究科

やり直しのための信号数学 第22回



DCTによる信号処理応用(その1)

三谷 政昭

前回(2004年3月号)は「DCTの高速計算アルゴリズム」と題し，FFT(高速フーリエ変換)を利用する方法とDCTの係数行列をスパースな(0の要素が多い)行列に分解する方法をとりあげて，基本的な考えかたを中心に解説した．
今回からは，“実務に直結して応用できるDCTアプリケーション”として，いろいろな信号データ処理にDCTを適用する事例をとりあげる．まずは，1次元信号(主として，音声)を題材に，雑音の除去，好みの音の生成(グラフィック・イコライザ)，プッシュ・ホンの電話番号の送出，選択，認識などへのDCTの適用のしかた，考えかたを中心にわかりやすく説明する．(筆者)

パーシバルの定理

基本的には，DFT(ディジタル・フーリエ変換)とDCTとの間には密接な関係があり，DCTとDST(ディジタル・サイン変換)を組み合わせることによってDFT値を求めることができる(詳細は本連載の2003年5月号の第16回「DFTからDCTへの橋渡し」および8月号の第17回「DCTによる信号解析の基礎」を参照)．したがって，従来のDFT，IDFTに基づいた種々の信号解析や信号処理アプリケーションは，すべてDCT，IDCTを利用して実現可能であるといえる．

たとえば，不規則な信号のもつ統計的性質を表す手段として，確率密度関数がある．この確率密度関数において，横軸を周波数f[Hz](あるいは角周波数$\omega = 2\pi f$[rad/秒])にしたものは，電力スペクトル密度関数(あるいはパワー・スペクトル密度関数; power spectrum density function)と呼ばれている．

いま，ディジタル信号波形$\{x_k\}_{k=0}^{k=N-1}$を考えれば，DCT値$\{C_\ell\}_{\ell=0}^{\ell=N-1}$，すなわち，

$$C_\ell = \frac{1}{N}\sum_{k=0}^{N-1}\gamma_\ell x_k \cos\left\{\frac{(2k+1)\ell}{2N}\pi\right\} \quad \cdots\cdots (1)$$

ただし，$\gamma_\ell = \begin{cases} 1 ; & \ell = 0 \\ \sqrt{2} ; & \ell \neq 0 \end{cases}$

に対し，電力スペクトル密度$\{P_\ell^{(XX)}\}_{\ell=0}^{\ell=N-1}$は，

$$P_\ell^{(XX)} = (C_\ell)^2 \quad ; \quad \ell = 0, 1, 2, \cdots, (N-1) \cdots\cdots (2)$$

と表される．ここで，添え字のℓは周波数に対応し，周波数分解能Δf[Hz]が，

$$\Delta f = \frac{f_s}{2N} \quad ; \ f_s \text{はサンプリング周波数} \cdots\cdots (3)$$

であることから，

$$\ell \Rightarrow \ell\,\Delta f\,[\mathrm{Hz}] \quad \cdots\cdots (4)$$

を表すことになる．

ところで，電力スペクトル密度に関係する「パーシバルの定理(あるいはパーシバルの等式)」をいま一度思い起こしていただきたい．すなわち，

『1サンプルあたりの平均電力Pは，電力スペクトル密度$(C_\ell)^2$の総和に等しい』

といわれる性質であり，

$$P = \frac{1}{N}\sum_{k=0}^{N-1}(x_k)^2 = \sum_{\ell=0}^{N-1}(C_\ell)^2 \quad \cdots\cdots (5)$$

という数式で表される(詳細は，2002年1月号の第6回「DFTによる基本的な信号分析」を参照)．つまり，ディジタル信号波形の時間領域で算出した平均電力，

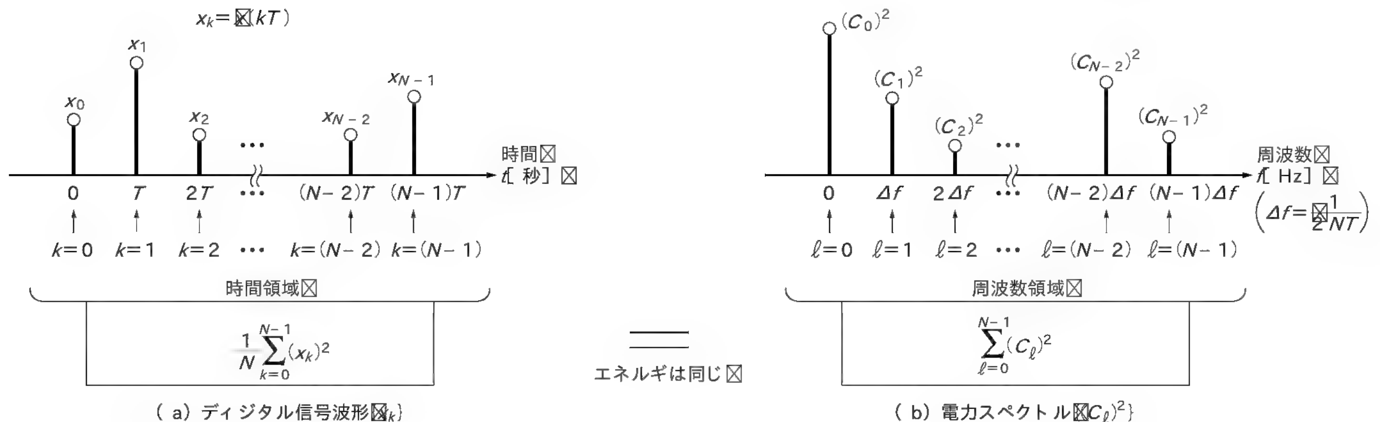
$$P\left(= \frac{1}{N}\sum_{k=0}^{N-1}(x_k)^2\right) \quad \cdots\cdots (6)$$

は，周波数領域で算出した電力スペクトル密度$(C_\ell)^2$[$\ell = 0, 1, 2, \cdots, (N-1)$]の総和，

$$P\left(= \sum_{\ell=0}^{N-1}(C_\ell)^2\right) \quad \cdots\cdots (7)$$

で表される以外の，どこかわからぬところに消えてしまったりはしないということである(図22.1)．





(a) ディジタル信号波形 $\{x_k\}$　(b) 電力スペクトル $\{(C_\ell)^2\}$

図22.1　パーシバルの定理

例題1

図22.2のディジタル信号波形 $\{x_k\}_{k=0}^{k=2}$ のDCT値を計算して得られた結果から，式(5)のパーシバルの定理が成り立つことを検証せよ．

解答1

まず，1サンプルあたりの平均電力 P は式(6)に基づき，信号値($x_0 = 2$, $x_1 = -4$, $x_2 = 5$)を代入して，

$$P = \frac{1}{3}\{2^2 + (-4)^2 + 5^2\} = 15 \quad \cdots\cdots(8)$$

と求まる．

さて，図22.2より，サンプリング間隔 $T = 1/30$[秒]なので，サンプリング周波数 f_s[Hz]は，

$$f_s = \frac{1}{T} = \frac{1}{(1/30)} = 30\ [\mathrm{Hz}]$$

である．また，$N = 3$ サンプルに対するDCTの周波数分解能 Δf は，式(3)より，

$$\Delta f = \frac{f_s}{2N} = \frac{30}{2 \times 3} = 5\ [\mathrm{Hz}]$$

となり，DCT値 $\{C_\ell\}_{\ell=0}^{\ell=2}$ はそれぞれ，

C_0: 0[Hz],　C_1: 5[Hz],　C_2: 10[Hz]

の周波数に対応するものを表すことに注意してもらいたい．

一方，$N = 3$ サンプルに対するDCT値は，式(1)より，

$$\begin{cases} C_0 = \frac{1}{3}(x_0 + x_1 + x_2) \\ C_1 = \frac{1}{3}\left\{\sqrt{2}x_0\cos\left(\frac{\pi}{6}\right) + \sqrt{2}x_1\cos\left(\frac{3\pi}{6}\right) + \sqrt{2}x_2\cos\left(\frac{5\pi}{6}\right)\right\} \\ C_2 = \frac{1}{3}\left\{\sqrt{2}x_0\cos\left(\frac{2\pi}{6}\right) + \sqrt{2}x_1\cos\left(\frac{6\pi}{6}\right) + \sqrt{2}x_2\cos\left(\frac{10\pi}{6}\right)\right\} \end{cases}$$

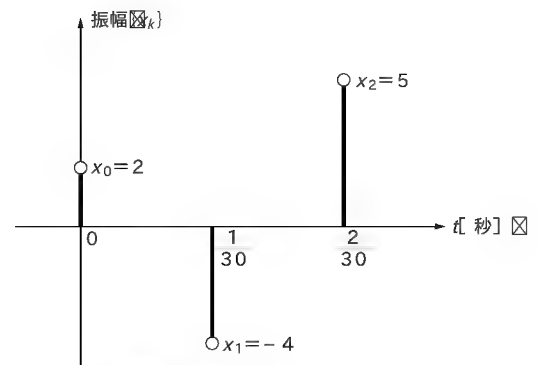


図22.2　例題1 ディジタル信号波形

となる関係が得られ，

$$\begin{cases} C_0 = \frac{1}{3}(x_0 + x_1 + x_2) \\ C_1 = \frac{\sqrt{6}}{6}(x_0 - x_2) \\ C_2 = \frac{\sqrt{2}}{6}(x_0 - 2x_1 + x_2) \end{cases} \quad \cdots\cdots(9)$$

と整理される．そこで，信号値($x_0 = 2$, $x_1 = -4$, $x_2 = 5$)を式(9)に代入してDCT値を計算する．

$$\begin{cases} C_0 = \frac{1}{3}(2 - 4 + 5) = 1 \\ C_1 = \frac{\sqrt{6}}{6}(2 - 5) = -\frac{\sqrt{6}}{2} \\ C_2 = \frac{\sqrt{2}}{6}\{2 - 2 \times (-4) + 5\} = \frac{5\sqrt{2}}{2} \end{cases} \quad \cdots\cdots(10)$$

この結果を式(7)に代入すれば，

$$P = 1^2 + \left(-\frac{\sqrt{6}}{2}\right)^2 + \left(\frac{5\sqrt{2}}{2}\right)^2$$
$$= 1 + \frac{6}{4} + \frac{50}{4} = 15 \quad \cdots\cdots(11)$$

と求められる．よって，式(8)と同じ値になるので，式(5)の妥当性が検証できたのである．

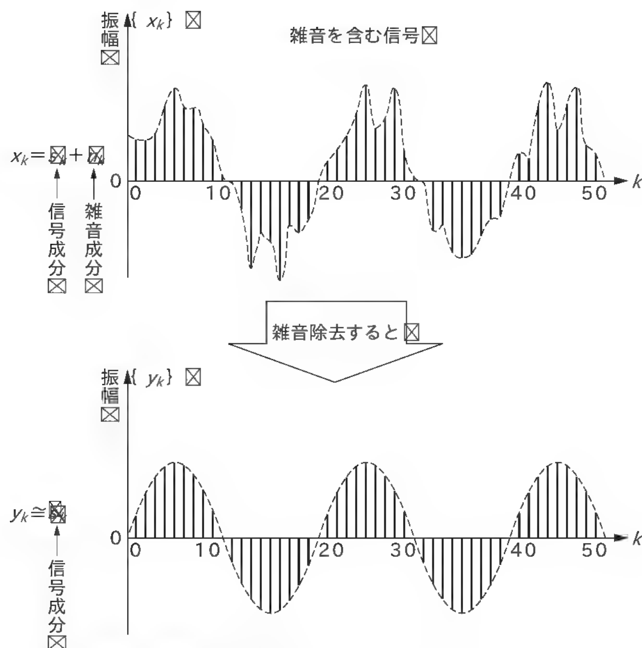


図 22.3　雑音を除去する処理

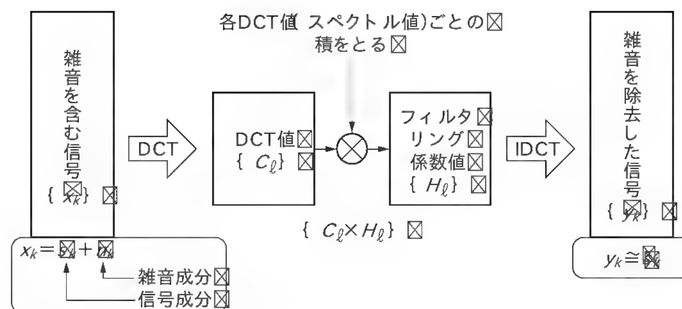


図 22.4　DCT による雑音の除去(フィルタリング)プロセス

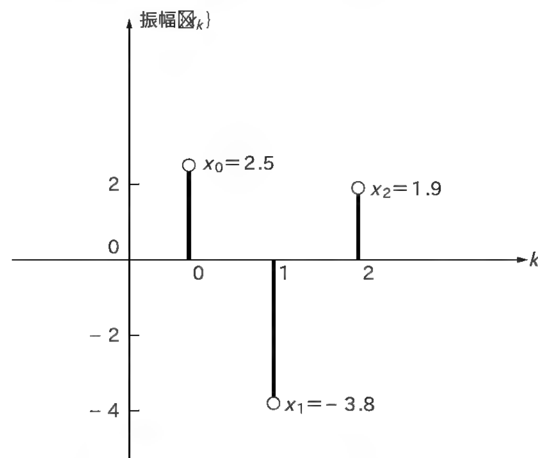


図 22.5　例題2 ディジタル信号波形

雑音を除去する処理

いま，雑音を含む信号 $\{x_k\}_{k=0}^{k=N-1}$，

$$x_k = s_k + n_k \quad \cdots\cdots(12)$$

ただし，s_k：信号成分，n_k：雑音成分

から信号成分を取り出す処理において，DCT 計算を利用することを考えてみよう(図 22.3)．

基本的には，雑音を含む信号 $\{x_k\}_{k=0}^{k=N-1}$ を DCT 処理して周波数成分を分析し，その周波数スペクトル値 $\{C_\ell\}_{\ell=0}^{\ell=N-1}$ に対して，信号成分と思われるものには"1"，雑音成分と思われるものには"0"を掛けて，さらに IDCT 処理するのである．つまり，信号は雑音よりスペクトル値が大きいことを利用し，信号の周波数スペクトルには"1"を掛けて取り出し，不要な雑音には"0"を掛けて取り除くという構図である(図 22.4)．このとき，DCT と IDCT の計算に，高速計算アルゴリズムを利用できる．

以下に，雑音除去の処理手順について，具体的に示しておこう．

手順1　DCT による周波数成分の計算

雑音を含む信号 $\{x_k\}_{k=0}^{k=N-1}$ を，式(1)に基づき，DCT 計算し，その周波数成分 $\{C_\ell\}_{\ell=0}^{\ell=N-1}$ を求める．

手順2　信号と雑音の識別判定

周波数成分 $\{C_\ell\}_{\ell=0}^{\ell=N-1}$ の値の大小を見て，信号と雑音の識別を行う．たとえば，

$$\begin{cases} |C_\ell| \geqq \varepsilon \text{であれば，} C_\ell \text{は信号成分} \\ |C_\ell| < \varepsilon \text{であれば，} C_\ell \text{は雑音成分} \end{cases} \quad \cdots\cdots(13)$$

とすればよい．ここで，ε は雑音と信号とを切り分けるための判定レベル(しきい値)であり，適切に定めておく必要がある．

手順3　雑音除去の計算

周波数成分 $\{C_\ell\}_{\ell=0}^{\ell=N-1}$ に掛ける係数を $\{H_\ell\}_{\ell=0}^{\ell=N-1}$ とするとき，

$$\begin{cases} \cdot\ \text{信号成分に対しては} H_\ell=1 \\ \cdot\ \text{雑音成分に対しては} H_\ell=0 \end{cases} \quad \cdots\cdots(14)$$

として，

$$G_\ell = C_\ell \times H_\ell \quad \cdots\cdots(15)$$

を計算して，雑音を除去した周波数スペクトル特性 $\{G_\ell\}_{\ell=0}^{\ell=N-1}$ を作成する．

手順4　IDCT による時間波形の再合成

手順2で得られた信号のみの周波数成分 $\{G_\ell\}_{\ell=0}^{\ell=N-1}$ を有する信号 $\{y_k\}_{k=0}^{k=N-1}$ を再合成するために，

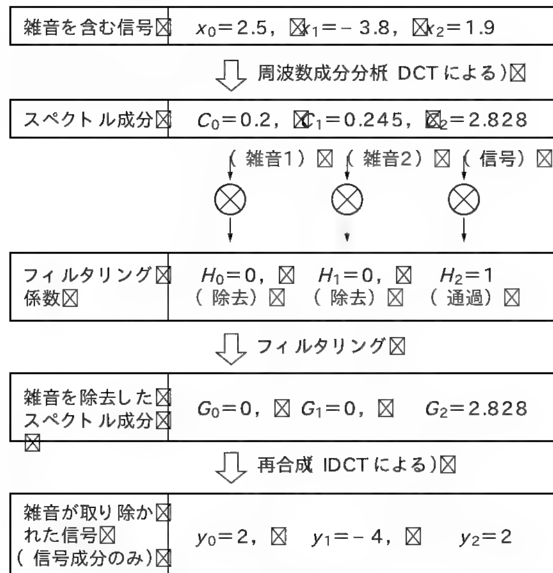
$$y_k = \sum_{\ell=0}^{N-1} \gamma_\ell G_\ell \cos\left\{\frac{(2k+1)\ell}{2N}\pi\right\} \quad \cdots\cdots(16)$$

$$\text{ただし，} \gamma_\ell = \begin{cases} 1 & ; \ell = 0 \\ \sqrt{2} & ; \ell \neq 0 \end{cases}$$

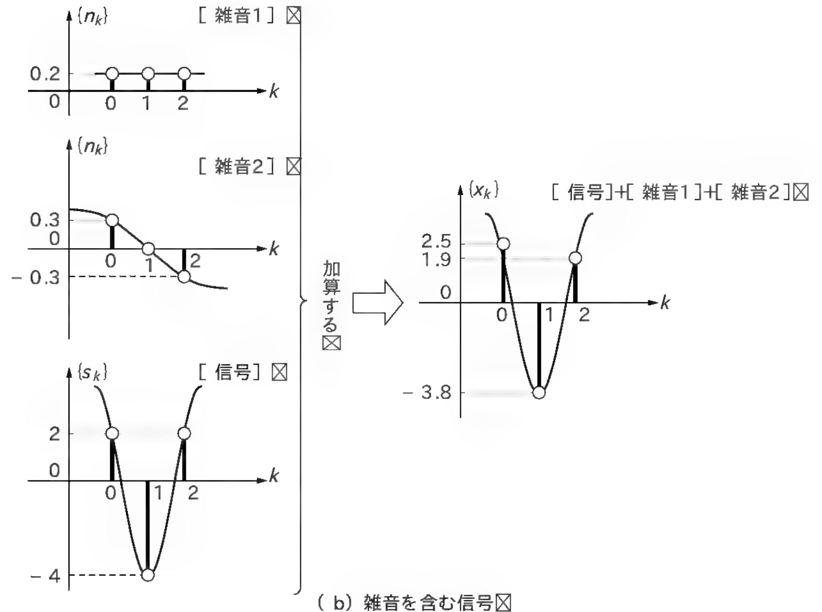
に基づき，IDCT 計算する．

以上の手順を踏んで，DCT 値と IDCT 値を算出することに





(a) 計算結果



(b) 雑音を含む信号

図22.6 例題2 のフィルタリング処理

より，ディジタル雑音除去システムが実現できるという流れである．

例題2

図22.5に示す雑音を含んだ信号から，DCTとIDCTを利用して雑音を除去したい．手順1～手順4に基づき，フィルタリング処理するようすについて信号値を計算し，確認せよ．ただし，信号と雑音を識別する判定レベル ε は0.8とする．

解答2

以下に，手順ごとに信号値の計算結果を示す(図22.6)．

手順1 周波数成分(DCT値)の計算

信号値($x_0=2.5$, $x_1=-3.8$, $x_2=1.9$)を式(9)に基づき，DCT処理して周波数成分を求める．

$$\begin{cases} C_0=\dfrac{1}{3}(2.5-3.8+1.9)=0.2 \\ C_1=\dfrac{\sqrt{6}}{6}(2.5-1.9)=\dfrac{\sqrt{6}}{10}\fallingdotseq 0.245 \\ C_2=\dfrac{\sqrt{2}}{6}\{2.5-2\times(-3.8)+1.9\}=2\sqrt{2}\fallingdotseq 2.828 \end{cases} \quad \cdots\cdots(17)$$

手順2 信号と雑音の識別判定

手順1の結果から，判定レベル $\varepsilon=0.8$ なので，式(13)に基づき，信号と雑音を切り分ける．

$$\begin{cases} C_0=0.2<0.8 \Rightarrow \text{雑音成分} \\ C_1\fallingdotseq 0.245<0.8 \Rightarrow \text{雑音成分} \\ C_2\fallingdotseq 2.828\geq 0.8 \Rightarrow \text{信号成分} \end{cases} \quad \cdots\cdots(18)$$

手順3 雑音除去の計算

各周波数成分$\{C_0, C_1, C_2\}$ごとに掛けるフィルタリング係数値は，

$$\begin{cases} H_0=0\text{(雑音なので，除去する)} \\ H_1=0\text{(雑音なので，除去する)} \\ H_2=1\text{(信号なので，通す)} \end{cases}$$

とすればよい．その結果，雑音が取り除かれた信号の周波数成分は以下のようになる．

$$\begin{aligned} &\{G_0, G_1, G_2\} \\ &=\{C_0\times H_0, C_1\times H_1, C_2\times H_2\} \quad \cdots\cdots(19) \\ &=\{0, 0, 2\sqrt{2}\} \end{aligned}$$

手順4 IDCTによる時間波形の再合成

手順3で得られた周波数成分$\{G_0, G_1, G_2\}$を有する信号を再合成するために，式(16)に基づき，サンプル数 $N=3$ として IDCT値を計算する．すなわち，

$$\begin{cases} y_0=G_0+\sqrt{2}G_1\cos\left(\dfrac{\pi}{6}\right)+\sqrt{2}G_2\cos\left(\dfrac{2\pi}{6}\right) \\ y_1=G_0+\sqrt{2}G_1\cos\left(\dfrac{3\pi}{6}\right)+\sqrt{2}G_2\cos\left(\dfrac{6\pi}{6}\right) \\ y_2=G_0+\sqrt{2}G_1\cos\left(\dfrac{5\pi}{6}\right)+\sqrt{2}G_2\cos\left(\dfrac{10\pi}{6}\right) \end{cases}$$

となる関係より，

$$\begin{cases} y_0=G_0+\dfrac{\sqrt{6}}{2}G_1+\dfrac{\sqrt{2}}{2}G_2 \\ y_1=G_0-\sqrt{2}G_2 \\ y_2=G_0-\dfrac{\sqrt{6}}{2}G_1+\dfrac{\sqrt{2}}{2}G_2 \end{cases} \quad \cdots\cdots(20)$$

と整理できる．式(19)の値$\{G_0, G_1, G_2\}$を式(20)に代入し，出力信号$\{y_0, y_1, y_2\}$を計算する．

$$\begin{cases} y_0 = 0 + \dfrac{\sqrt{6}}{2} \times 0 + \dfrac{\sqrt{2}}{2} \times 2\sqrt{2} = 2 \\ y_1 = 0 - \sqrt{2} \times 2\sqrt{2} = -4 \\ y_2 = 0 - \dfrac{\sqrt{6}}{2} \times 0 + \dfrac{\sqrt{2}}{2} \times 2\sqrt{2} = 2 \end{cases} \quad \cdots\cdots (21)$$

以上より，$y_k = s_k$なので，雑音を除去できたことになる．

例題3

図22.7の観測信号データに埋もれているモータの回転音(単一周波数で250[Hz])を，DCTとIDCTを利用して取り出したい．手順1～手順4に基づき，計算の流れと処理結果を示せ．

解答3

例題2とほぼ同様の処理の流れでよいが，250[Hz]がDCT処理した値のどれに当たるのかを見極める必要がある．図22.7より，サンプリング間隔T= 0.001[秒]であり，逆数をとってサンプリング周波数は$1/T$= 1/0.001= 1000[Hz]であることがわかる．さらに，周波数サンプル数はN= 4で，式(3)より周波数分解能は1000[Hz]/8= 125[Hz]であることから，250[Hz]は250[Hz]/125[Hz]= 2(= ℓ)となる．図22.8に，計算結果のみを示しておくので，各自で検算してほしい．

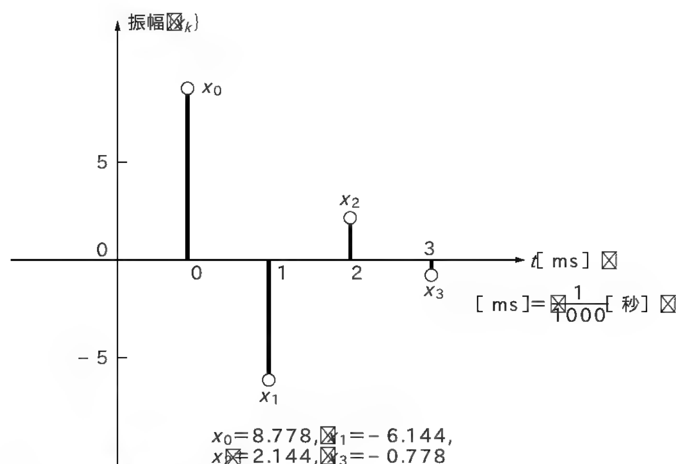


図22.7 **例題3** 観測信号データ

雑音を含む モータ音の信号 $\{x_k\}_{k=0}^{k=3}$	図22.7参照
⇩ 周波数成分分析(DCTによる)	
スペクトル成分 $\{C_\ell\}_{\ell=0}^{\ell=3}$	C_0(=0[Hz]，直流)=1 C_1(=125[Hz])=2 C_2(=250[Hz])=3(モータ音に相当) C_3(=375[Hz])=4
⇩ モータ音に相当するスペクトル成分の抽出	
モータ音の スペクトル成分 $\{G_\ell\}_{\ell=0}^{\ell=3}$	G_0=G_1=G_3=0 (モータ音以外は"0"にする) G_2=3(モータ音)
⇩ 再合成(IDCTによる)	
モータ音 $\{y_k\}_{k=0}^{k=3}$	y_0=3，y_1=−3，y_2=−3，y_3=3

図22.8 **例題3** のモータ音の抽出処理

好みの音の生成

カー・オーディオや卓上ステレオなどでは，必ずといってよいほど音質の調整ができるようになっている．とくに周波数ごとに調整できるようなものもあり，グラフィック・イコライザと呼ばれて車の中で聞きやすい音や個人の好みに合わせた音作りができる．そこで，こうした好みの音作りのDCTバージョンを作ってみようというわけである．基本的なしくみは前述の雑音除去の考えかたを拡張するだけで簡単にできてしまうので，まったくもって驚きである(図22.9)．

手順1　DCTによる周波数成分の計算

式(1)に基づき，音声信号$\{x_k\}_{k=0}^{k=N-1}$をDCTして，その周波数成分$\{C_\ell\}_{\ell=0}^{\ell=N-1}$を求める．

手順2　聞かせたい音域のボリューム調整

周波数成分$\{C_\ell\}_{\ell=0}^{\ell=N-1}$に掛ける係数を$\{H_\ell\}_{\ell=0}^{\ell=N-1}$とするとき

・低音や高音などの聞かせたい周波数ℓに対しては，H_ℓを1より大きい値
・低音や高音などの聞かせたくない周波数ℓに対しては，H_ℓを1より小さい値
　……… (22)

に設定し，

$$G_\ell = C_\ell \times H_\ell \quad \cdots\cdots (23)$$

を計算して，好みの音の周波数スペクトル特性$\{G_\ell\}_{\ell=0}^{\ell=N-1}$を作成する．このとき，$\{H_\ell\}_{\ell=0}^{\ell=N-1}$が音質を調整するためのパラメータである．

手順3　IDCTによる時間波形の再合成

手順2で得られた周波数成分$\{G_\ell\}_{\ell=0}^{\ell=N-1}$を有する信号を，式(16)に基づき，再合成する．

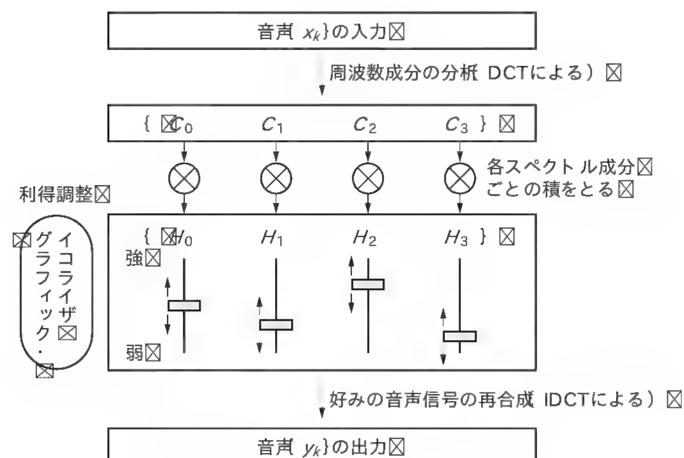


図22.9 DCT型グラフィック・イコライザの簡単なモデル



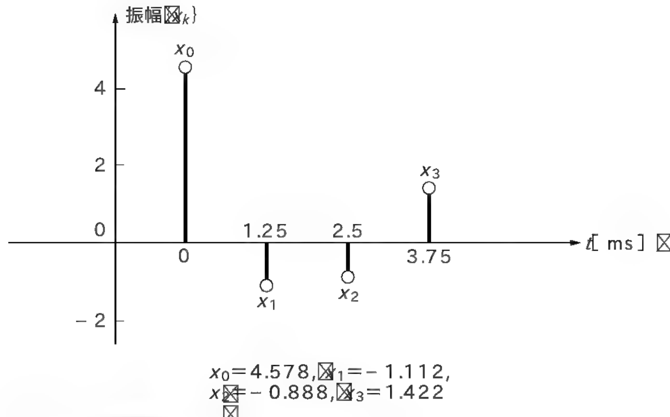


図22.10 例題4 音楽データ

例題4

CDに録音された音楽データがあるとして，グラフィック・イコライザで処理してシャカシャカという高音を強めるようにし，低音を少し弱めた音として聴きたい．そこで，図22.10に示す音楽データの周波数成分を分析して，直流分は完全に除去し，100[Hz]の低音成分は0.5倍に，200[Hz]の中音成分は1.5倍に，300[Hz]の高音成分は2倍にすることを考える．手順1～手順3に基づき，グラフィック・イコライザを通した後の信号値を示せ．

解答4

まず，図22.10より，サンプリング間隔 $T=1.25$[ms]であり，逆数を採ってサンプリング周波数は $1/T=1/0.00125=800$[Hz]であることがわかる．周波数サンプル数は $N=4$ で，式(3)より周波数分解能は800[Hz]/8=100[Hz]であることから，$\ell=1$ は100[Hz]，$\ell=2$ は200[Hz]，$\ell=3$ は300[Hz]となる．

例題3 のモータ回転音の抽出の例とほぼ同じ処理であり，手順3のフィルタリング係数 $\{H_0, H_1, H_2, H_3\}$ を，音質調整の利得係数と読み換えることによりグラフィック・イコライザをDCT計算で実現できるのである．題意より，直流分は完全に0にすればよいので $H_0=0$，以下同様に $H_1=0.5$，$H_2=1.5$，$H_3=2$ と設定する．図22.11に，計算結果のみを示しておくので，必ず検算してもらいたい．

また，同様の手順により，トランペット，ピアノ，ギター，ヴァイオリンなどの楽器音，“あ”，“い”，…などの母音を生成することも可能である．基本的には，DCT計算により楽器音や母音の周波数成分を分析して，IDCT計算により同一の音を作り出す(再合成する)のである．

プッシュ・ホンの番号送出・選択・認識

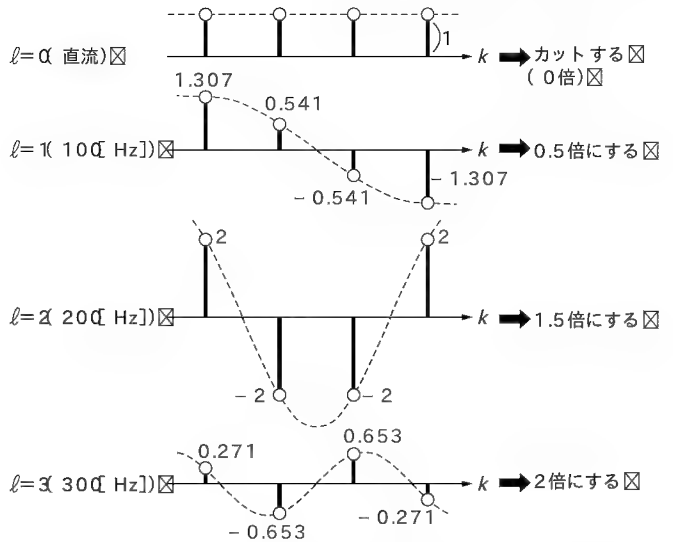
みなさんがおなじみのプッシュ・ホンは，それ以前の黒電話の回転ダイヤルに代わって，数字ボタンを押すだけでよく，こ

音楽データ $\{x_k\}_{k=0}^{k=3}$	4.578	−1.112	−0.888	1.422

⇩ 周波数成分分析(DCTによる)

スペクトル成分 $\{C_\ell\}_{\ell=0}^{\ell=3}$	1	1	2	0.5

|| 周波数ごとの時間波形に分離してみると



⇩ グラフィック・イコライザの利得調整

利得調整後のスペクトル成分 $\{G_\ell\}_{\ell=0}^{\ell=3}$	1×0	1×0.5	2×1.5	0.5×2
	0	0.5	3	1

⇩ 再合成(IDCTによる)

好みの音の信号 $\{y_k\}_{k=0}^{k=3}$	4.194	−4.036	−1.964	1.806

|| 周波数ごとの時間波形に分離してみると

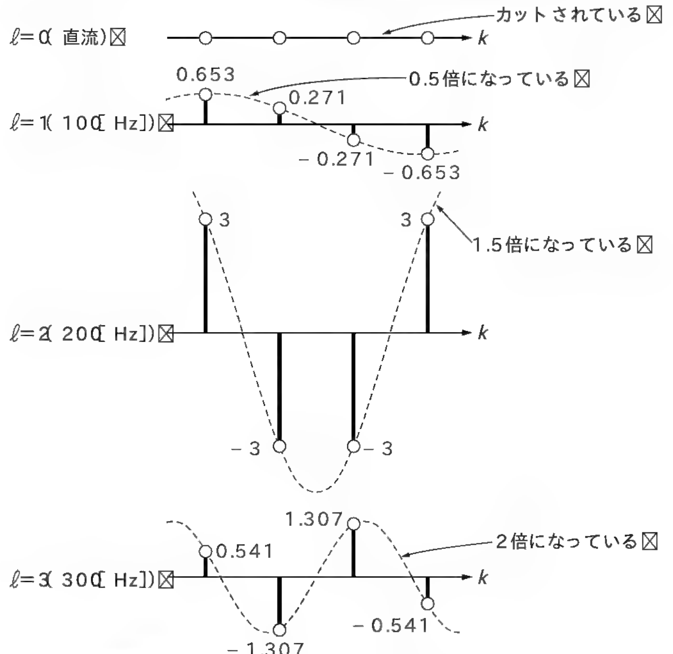


図22.11 解答4 のグラフィック・イコライザの処理

れによって特定の周波数の音(トーン信号という)を発信し，これが交換機を作動させて，通信相手にスムーズに接続されるしくみになっている．つまり,「ピッ，ポッ，パッ」という音で電話番号の数字の情報を送り出し，音を受けて受信した数字を分析・認識しているのである(図22.12)．ここでは，電話番号に対応するトーン信号の送出，認識のしくみをDCT，IDCTで作成して，電話が音でつながる原理を理解してもらうことにする．

いま，1から7までの7種類の数字と$N=3$サンプルのトーン信号波形とを，図22.13のように対応させることにする．このとき，式(9)を利用して各トーン信号のDCT値$\left\{C_\ell\right\}_{\ell=0}^{\ell=2}$を求めると，次のようになる．

$$\begin{cases} \text{“1”} \Leftrightarrow C_0=1,\ C_1=0,\ C_2=0 \\ \text{“2”} \Leftrightarrow C_0=0,\ C_1=\sqrt{6},\ C_2=0 \\ \text{“3”} \Leftrightarrow C_0=0,\ C_1=0,\ C_2=\sqrt{2} \\ \text{“4”} \Leftrightarrow C_0=1,\ C_1=\sqrt{6},\ C_2=0 \\ \text{“5”} \Leftrightarrow C_0=1,\ C_1=0,\ C_2=\sqrt{2} \\ \text{“6”} \Leftrightarrow C_0=0,\ C_1=\sqrt{6},\ C_2=\sqrt{2} \\ \text{“7”} \Leftrightarrow C_0=1,\ C_1=\sqrt{6},\ C_2=\sqrt{2} \end{cases} \quad \cdots\cdots\cdots\cdots (24)$$

このように，トーン信号波形に含まれる周波数成分の組み合わせにより7種類の数字を区別するところに,「ピッ，ポッ，パッ」音による番号識別のアイデアが潜ませてあるわけである．

以上より，トーン信号による電話番号の送出，識別，認識はDCTとIDCTを用いて図22.14のように構成すれば実現可能なことが，容易に類推できるであろう．

例題5

いま，図22.14の電話番号の送受信モデルにおいて，相手先の電話番号が“175”とするとき，送出されるトーン信号波形$\left\{x_k\right\}_{k=0}^{k=8}$を求めよ．

解答5

式(24)に基づき，電話番号に対応するDCT値$\left\{C_\ell\right\}_{\ell=0}^{\ell=2}$からIDCTを計算することにより，3サンプルごとに送出されるトーン信号波形$\left\{x_k\right\}_{k=0}^{k=2}$を求めればよい．たとえば，電話番号の最初の数字“1”であれば，式(24)より，

$\{C_0=1,\ C_1=0,\ C_2=0\}$

であり，$G_\ell=C_\ell$として式(20)に代入して得られる値(y_kをx_kに読み換える)がトーン信号波形となる．

$\{x_0=1,\ x_1=1,\ x_2=1\}$

ほかも同様に計算できるので確認してもらいたい．

例題6

いま，図22.14の電話番号の送受信モデルにおいて，受信されたトーン信号波形$\left\{x_k\right\}_{k=0}^{k=8}$が図22.15であるとき，送信元の電話番号を求めよ．

解答6

3サンプルごとのトーン信号をDCT計算して，周波数成分の



周波数[Hz]		高い周波数のグループ		
		1209	1336	1477
低い周波数のグループ	697	1	2	3
	720	4	5	6
	852	7	8	9
	941	*	0	#

図22.12　プッシュ・ホンの電話番号とトーン信号

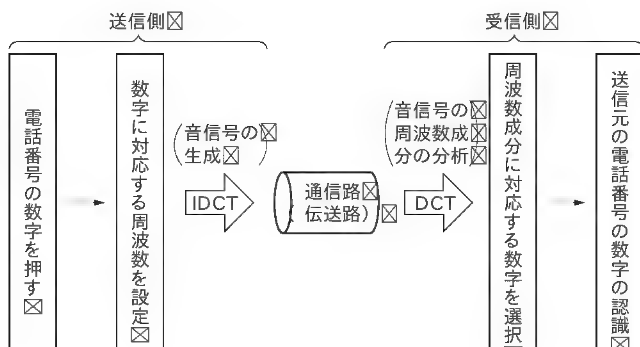


図22.14　DCT，IDCTによる電話番号の送受信モデル

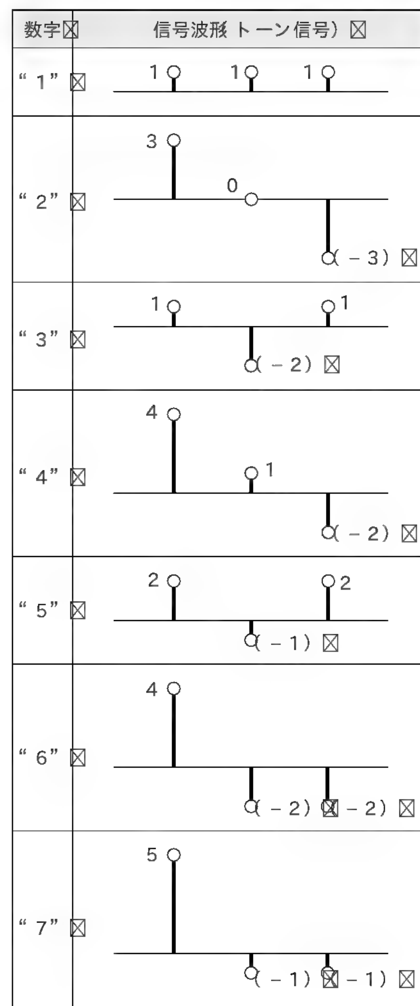


図22.13
電話番号の数字と
信号波形の対応



組み合わせを調べ，式(24)に基づき，DCT 値に対応する電話番号の数字を確定すればよい．たとえば，トーン信号波形の最初の3個のサンプル値が，

$\{x_0 = 3,\ x_1 = 0,\ x_2 = -3\}$

であれば，式(9)より DCT 値は，

$\{C_0 = 0,\ C_1 = \sqrt{6},\ C_2 = 0\}$

となり，式(24)より数字"2"であることがわかる．

ほかも同様の計算で求められるので検証しておいてほしい．

オーバ・ラップ（重なり）データの DCT 値の効率的計算法

ここでは，サンプル数 N[個]のディジタル信号波形 $\{x_k\}_{k=0}^{k=N-1}$ に対して算出した DCT 値 $\{C_\ell\}_{\ell=0}^{\ell=N-1}$ を用いて，m[個]分のサンプルだけずらした N サンプルの信号波形に対する DCT 値を，計算量が少なくて効率よく求める方法を紹介する(図22.16)．

いま，たとえば $(N-m)$[個]のサンプルがオーバ・ラップする(重なり)信号波形 $\{\hat{x}_k = x_{k+m}\}_{k=0}^{k=N-1}$ に対する DCT 値 $\{G_\ell^{(m)}\}_{\ell=0}^{\ell=N-1}$ は，式(1)に基づき，

$$G_\ell^{(m)} = \frac{1}{N}\sum_{k=0}^{N-1}\gamma_\ell \hat{x}_k \cos\left\{\frac{(2k+1)\ell}{2N}\pi\right\} = \frac{1}{N}\sum_{k=0}^{N-1}\gamma_\ell x_{k+m} \cos\left\{\frac{(2k+1)\ell}{2N}\pi\right\} \quad \cdots\cdots(25)$$

であり，$n = k + m$（あるいは $k = n - m$）とおけば，

$$G_\ell^{(m)} = \frac{1}{N}\sum_{n=m}^{N-1+m}\gamma_\ell x_n \cos\left\{\frac{(2n-2m+1)\ell}{2N}\pi\right\} \quad \cdots\cdots(26)$$

のように式が変形される．そこで，cos の{ }の中の計算式において，

$$2n - 2m + 1 = (2n+1) - 2m \quad \cdots\cdots(27)$$

と考え，さらに三角関数の加法定理のうち，

$$\cos(\alpha - \beta) = \cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\beta) \quad \cdots\cdots(28)$$

となる関係を適用する．また，

$$\frac{(2n+1)\ell}{2N}\pi = \alpha \quad \cdots\cdots(29)$$

$$\frac{2m\ell}{2N}\pi = \frac{m\ell}{N}\pi = \beta \quad \cdots\cdots(30)$$

と置けば，

$$\frac{(2n-2m+1)\ell}{2N}\pi = \alpha - \beta \quad \cdots\cdots(31)$$

と表されるので，式(26)は次のように変形される．

$$G_\ell^{(m)} = \frac{1}{N}\sum_{n=m}^{N-1+m}\gamma_\ell x_n\left[\cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\}\cos\left(\frac{m\ell}{N}\pi\right) + \sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}\sin\left(\frac{m\ell}{N}\pi\right)\right] \quad \cdots\cdots(32)$$

続けて，式(32)を整理すると，

$$G_\ell^{(m)} = \cos\left(\frac{m\ell}{N}\pi\right)\left[\frac{1}{N}\sum_{n=m}^{N-1+m}\gamma_\ell x_n \cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\}\right] + \sin\left(\frac{m\ell}{N}\pi\right)\left[\frac{1}{N}\sum_{n=m}^{N-1+m}\gamma_\ell x_n \sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}\right] \quad \cdots\cdots(33)$$

となる．さらに，[ ]の中の計算を次のように置き換えることを考える．

$$\begin{aligned}
&\frac{1}{N}\sum_{n=m}^{N-1+m}\gamma_\ell x_n \cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\} \\
&= \frac{1}{N}\sum_{n=0}^{m-1}\gamma_\ell(-x_n + x_n)\cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\} + \frac{1}{N}\sum_{n=m}^{N-1}\gamma_\ell x_n \cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\} + \frac{1}{N}\sum_{n=N}^{N-1+m}\gamma_\ell x_n \cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\} \\
&= -\frac{1}{N}\sum_{n=0}^{m-1}\gamma_\ell x_n \cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\} + \underbrace{\frac{1}{N}\sum_{n=0}^{N-1}\gamma_\ell x_n \cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\}}_{C_\ell} + \frac{1}{N}\sum_{n=N}^{N-1+m}\gamma_\ell x_n \cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\}
\end{aligned} \quad \cdots\cdots(34)$$

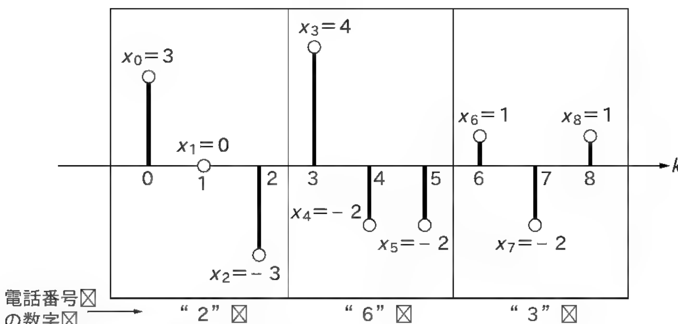


図22.15 例題6 トーン信号波形

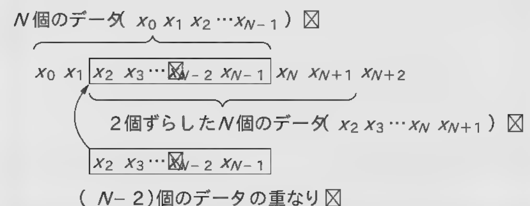


図22.16 オーバ・ラップのある二つのディジタル信号系列の例($m = 2$の場合)

$$\frac{1}{N}\sum_{n=m}^{N-1+m}\gamma_\ell x_n \sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}$$
$$=\frac{1}{N}\sum_{n=0}^{m-1}\gamma_\ell(-x_n+x_n)\sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}+\frac{1}{N}\sum_{n=m}^{N-1}\gamma_\ell x_n \sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}$$
$$+\frac{1}{N}\sum_{n=N}^{N-1+m}\gamma_\ell x_n \sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}$$
$$=-\frac{1}{N}\sum_{n=0}^{m-1}\gamma_\ell x_n \sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}+\underbrace{\frac{1}{N}\sum_{n=0}^{N-1}\gamma_\ell x_n \sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}}_{S_\ell}$$
$$+\frac{1}{N}\sum_{n=N}^{N-1+m}\gamma_\ell x_n \sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\} \quad\cdots\cdots(35)$$

式(34)と式(35)に基づき，式(33)は最終的に，

$$G_\ell^{(m)}=\cos\left(\frac{m\ell}{N}\pi\right)\left[C_\ell-\frac{1}{N}\sum_{n=0}^{m-1}\gamma_\ell x_n\cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\}+\frac{1}{N}\sum_{n=N}^{N-1+m}\gamma_\ell x_n\cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\}\right]$$
$$+\sin\left(\frac{m\ell}{N}\pi\right)\left[S_\ell-\frac{1}{N}\sum_{n=0}^{m-1}\gamma_\ell x_n\sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}+\frac{1}{N}\sum_{n=N}^{N-1+m}\gamma_\ell x_n\sin\left\{\frac{(2n+1)\ell}{2N}\pi\right\}\right]$$
$$\cdots\cdots(36)$$

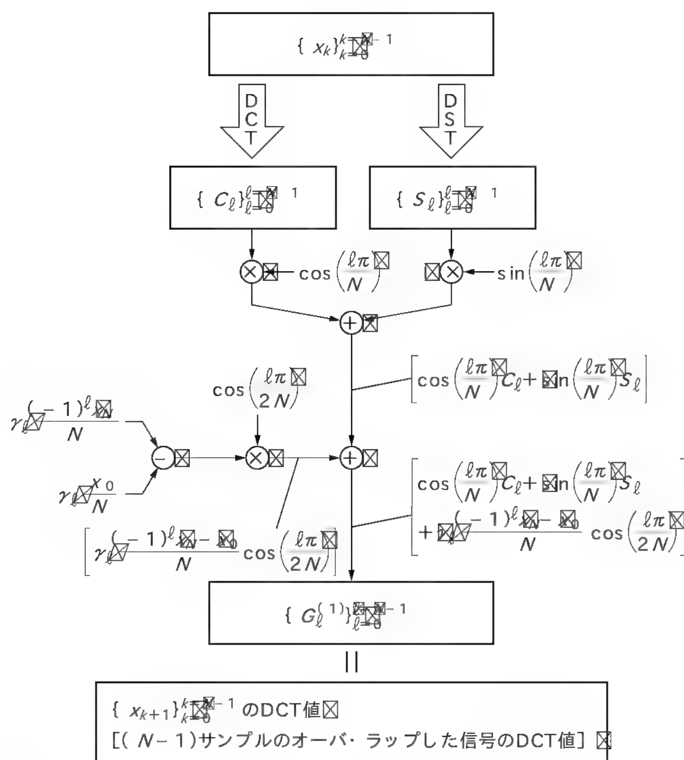


図22.17
オーバ・ラップした信号のDCT値の簡略計算プロセス($m=1$の場合)

と表される．ここで，C_ℓは式(1)のDCT値，S_ℓはDST値で，

$$S_\ell=\frac{1}{N}\sum_{k=0}^{N-1}\gamma_\ell x_k\sin\left\{\frac{(2k+1)\ell}{2N}\pi\right\} \quad\cdots\cdots(37)$$

$$\text{ただし，}\quad \gamma_\ell=\begin{cases}1 ; & \ell=0\\ \sqrt{2} ; & \ell\neq 0\end{cases}$$

で与えられる．

よって，式(36)より，オーバ・ラップした信号波形に対するDCT値$\{G_\ell^{(m)}\}_{\ell=0}^{\ell=N-1}$が，以前に求めた信号データに対するDCT値$\{C_\ell\}_{\ell=0}^{\ell=N-1}$とDST値$\{S_\ell\}_{\ell=0}^{\ell=N-1}$から求められることがわかる．つまり，オーバ・ラップした信号に対するDCT値計算を簡略化することができ，必要な計算のみを実行することにより高速なデータ処理を可能とする手法なのである．

たとえば，1サンプル移動したときの信号波形($m=1$に相当)に対するDCT値は，式(36)より，

$$G_\ell^{(1)}=\cos\left(\frac{\ell\pi}{N}\right)\left[C_\ell-\frac{1}{N}\gamma_\ell x_0\cos\left(\frac{\ell\pi}{2N}\right)+\frac{1}{N}\gamma_\ell x_N\cos\left\{\frac{(2N+1)\ell}{2N}\pi\right\}\right]$$
$$+\sin\left(\frac{\ell\pi}{N}\right)\left[S_\ell-\frac{1}{N}\gamma_\ell x_0\sin\left(\frac{\ell\pi}{2N}\right)+\frac{1}{N}\gamma_\ell x_N\sin\left\{\frac{(2N+1)\ell}{2N}\pi\right\}\right] \quad\cdots\cdots(38)$$

と計算される．ここで，

$$\frac{(2N+1)\ell}{2N}\pi=\ell\pi+\frac{\ell\pi}{2N} \;;\; \ell=0,\ 1,\ 2,\ \cdots,(N-1) \quad\cdots\cdots(39)$$

であることを考慮し，三角関数の性質，すなわち，

$$\begin{cases}\cos(\ell\pi+\theta)=(-1)^\ell\cos(\theta)\\ \sin(\ell\pi+\theta)=(-1)^\ell\sin(\theta)\end{cases} \quad\cdots\cdots(40)$$

となる関係において，$\theta=\ell\pi/2N$とすれば，式(38)は，

$$G_\ell^{(1)}=\cos\left(\frac{\ell\pi}{N}\right)C_\ell-\frac{1}{N}\gamma_\ell x_0\cos\left(\frac{\ell\pi}{2N}\right)\cos\left(\frac{\ell\pi}{N}\right)$$
$$+\frac{(-1)^\ell}{N}\gamma_\ell x_N\cos\left(\frac{\ell\pi}{N}\right)\cos\left(\frac{\ell\pi}{N}\right)$$
$$+\sin\left(\frac{\ell\pi}{N}\right)S_\ell-\frac{1}{N}\gamma_\ell x_0\sin\left(\frac{\ell\pi}{N}\right)\sin\left(\frac{\ell\pi}{2N}\right)$$
$$+\frac{(-1)^\ell}{N}\gamma_\ell x_N\sin\left(\frac{\ell\pi}{N}\right)\sin\left(\frac{\ell\pi}{2N}\right)$$
$$=\cos\left(\frac{\ell\pi}{N}\right)C_\ell+\sin\left(\frac{\ell\pi}{N}\right)S_\ell$$
$$+\gamma_\ell\frac{(-1)^\ell x_N-x_0}{N}\times\left[\cos\left(\frac{\ell\pi}{N}\right)\cos\left(\frac{\ell\pi}{2N}\right)+\sin\left(\frac{\ell\pi}{N}\right)\sin\left(\frac{\ell\pi}{2N}\right)\right] \quad\cdots\cdots(41)$$

と式変形される．さらに，式(41)の[　]内の三角関数の積和



式で $\alpha = \ell\pi/N$, $\beta = \ell\pi/2N$ として式(28)を適用すれば，1サンプル移動したときのDCT値 $G_\ell^{(1)}$ は，

$$G_\ell^{(1)} = \cos\left(\frac{\ell\pi}{N}\right)C_\ell + \sin\left(\frac{\ell\pi}{N}\right)S_\ell + \gamma_\ell \frac{(-1)^\ell x_N - x_0}{N}\cos\left(\frac{\ell\pi}{2N}\right) \quad \cdots\cdots(42)$$

で計算できる(図22.17)．

例題7

まず，図22.18(a)のディジタル信号 $\{x_k\}_{k=0}^{k=2}$ のDCT値を式(9)により直接求めた後，同図(b)の1サンプル遅れた信号 $\{x_k\}_{k=1}^{k=3}$ のDCT値を式(42)より計算せよ．

解答7

式(9)よりDCT値を求める．また，DST値は，式(37)より，$N=3$として，

$$\begin{cases} S_0 = 0 \\ S_1 = \frac{1}{3}\left\{\sqrt{2}x_0\sin\left(\frac{\pi}{6}\right) + \sqrt{2}x_1\sin\left(\frac{3\pi}{6}\right) + \sqrt{2}x_2\sin\left(\frac{5\pi}{6}\right)\right\} \\ S_2 = \frac{1}{3}\left\{\sqrt{2}x_0\sin\left(\frac{2\pi}{6}\right) + \sqrt{2}x_1\sin\left(\frac{6\pi}{6}\right) + \sqrt{2}x_2\sin\left(\frac{10\pi}{6}\right)\right\} \end{cases}$$

となる関係が得られ，

$$\begin{cases} S_0 = 0 \\ S_1 = \frac{\sqrt{2}}{6}(x_0 + 2x_1 + x_2) \\ S_2 = \frac{\sqrt{6}}{6}(x_0 - x_2) \end{cases} \quad \cdots\cdots(43)$$

で計算する．計算結果のみを以下に記しておくので，各自で検証してもらいたい．

●図22.18(a)のDCT値〔式(9)〕，DST値〔式(43)〕

$\{C_0 = 1,\ C_1 = \sqrt{6},\ C_2 = 0\}$ ……(44)

$\{S_0 = 0,\ S_1 = 2\sqrt{2}/3,\ S_2 = \sqrt{6}\}$ ……(45)

●図22.18(b)のDCT値

● 直接計算による算出値〔式(9)〕

$\{C_0 = 0,\ C_1 = 0,\ C_2 = \sqrt{2}\}$ ……(46)

● 効率的な計算法による算出値

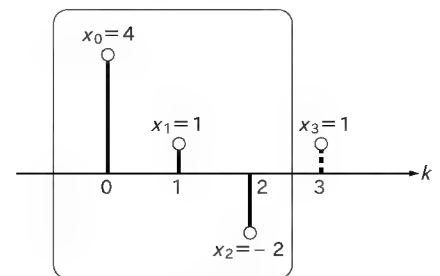
式(42)より，$N=3$として式(44)と式(45)を考慮すれば，以下のように求められる．

$\ell = 0$の場合

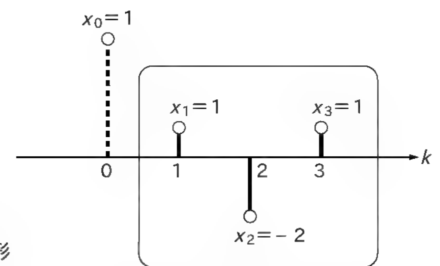
$$G_0^{(1)} = C_0 + S_0 + \frac{x_3 - x_0}{3} = 1 + 0 + \frac{1-4}{3} = 1 - 1 = 0 \quad \cdots\cdots(47)$$

$\ell = 1$の場合

$$G_1^{(1)} = \frac{1}{2}C_1 + \frac{\sqrt{3}}{2}S_1 + \sqrt{2}\times\frac{-x_3 - x_0}{3}\times\frac{\sqrt{3}}{2} = \frac{1}{2}\times\sqrt{6} + \frac{\sqrt{3}}{2}\times\frac{2\sqrt{2}}{3} + \sqrt{2}\times\frac{-1-4}{3}\times\frac{\sqrt{3}}{2} = \frac{\sqrt{6}}{2} + \frac{\sqrt{6}}{3} - \frac{5\sqrt{6}}{6} = 0 \quad \cdots\cdots(48)$$



(a) 原信号 $\{x_k\}_{k=0}^{k=2}$



(b) 2サンプルオーバーラップした信号 $\{x_k\}_{k=1}^{k=3}$

図22.18 **例題7** ディジタル信号波形

$\ell = 2$の場合

$$G_2^{(1)} = -\frac{1}{2}C_2 + \frac{\sqrt{3}}{2}S_2 + \sqrt{2}\times\frac{x_3 - x_0}{3}\times\frac{1}{2} = \frac{1}{2}\times 0 + \frac{\sqrt{3}}{2}\times\sqrt{6} + \sqrt{2}\times\frac{1-4}{3}\times\frac{1}{2} = \frac{3\sqrt{2}}{2} - \frac{\sqrt{2}}{2} = \sqrt{2} \quad \cdots\cdots(49)$$

以上より，式(47)～式(49)のDCT値が式(46)で直接計算した値に一致することを確認でき，式(42)によるオーバーラップした信号の周波数成分の効率的な計算法の正当性が検証された．

＊　　＊

次回は，“実務に直結して応用できるDCTアプリケーション”の第2弾として，2次元データの信号処理，すなわち画像処理を中心にわかりやすく解説する予定である．お楽しみに．

みたに・まさあき　東京電機大学工学部情報通信工学科

#「VxWORKS」を使ったRTOS技術の基礎と応用

第5回

VxWORKS TCP/IPプロトコル・スタックの設計と実装（後編）

濱口 遵一郎

前編では，組み込みシステムの開発プラットホームとしてのTornado/VxWORKSの特徴について解説しました．後編となる今回はVxWORKS TCP/IPプロトコル・スタックの設計と実装について詳しく説明します．

まずはじめに，VxWORKSネットワーキングのコンフィグレーションと初期化，そしてTCP/IPプロトコル・スタックの実体であるタスク`tNetTask`について，次にVxWORKS固有のアーキテクチャであるMUX/ENDとNPT（Network Protocol Toolkit 注1）のソフトウェア・モデルと`ipAttach`関数について，最後に`tNetTask`の送受信のデータ・パスを見ていきましょう．

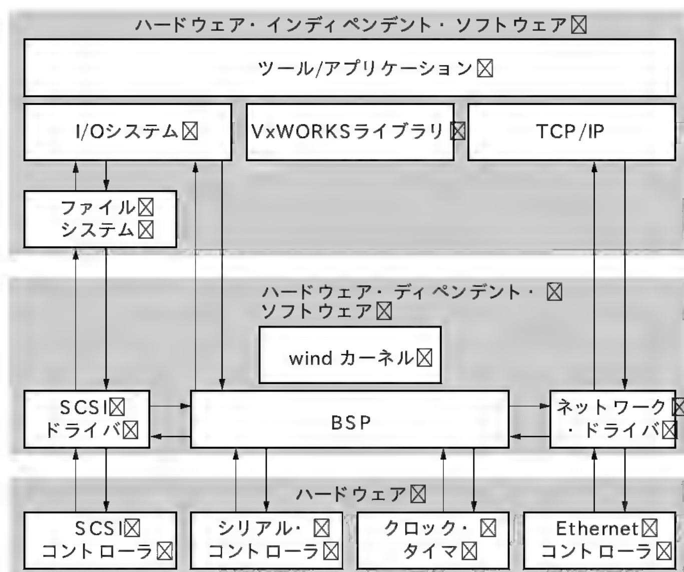


図1 VxWORKSのモジュール構成

解説の中で，VxWORKS TCP/IPプロトコル・スタックのベースとなったBSDとの違いも要所要所で言及していきます．前編とは異なり，後編は上級者向けです．

ターゲットとしてBSD TCP/IPプロトコル・スタックの実装に詳しい方か，何らかの形でVxWORKS TCP/IPプロトコル・スタックのソース・コードをトレースしたことがあり，より理解を深めたい方を想定しています．

VxWORKSネットワークがどう動いているか

ここでは，VxWORKSネットワークがどう動いているかを紹介します．まずは前編のおさらいとして，図1のVxWORKSのモジュール構成を見てみましょう．これによりVxWORKSがUNIX系OSやWindowsとどのくらい違うのかを再確認できます．

Ethernetなどのネットワーク・デバイスのハードウェアの初期化はEnhanced Network Driver 注2（以下END）とは分離して，Board Support Package 注3（以下BSP）の初期化の一環として行います．

ドライバ・ソフトウェアの初期化はENDで行います．さらに，VxWORKSではIEEE802（CSMA/CD）準拠のライブラリ`miiLib`が用意されており，Ethernetドライバをスクラッチで書くユーザに便宜を図っています．

● VxWORKSネットワーキングのコンフィグレーションと初期化

ユーザがVxWORKSネットワーキングのコンフィグレーションと初期化を行うには，Tornado Project Facility 注4という GUIによる方法と，コマンド・ラインからビルドを行う方法の2通りがあります．

注1：Network Protocol ToolkitはEND同様Wind Riverオリジナルのドライバ・モデルである．ENDがフレーム指向（リンク・レベルの情報をMUXが処理する）なのに対して，NPTはパケット指向（リンク・レベルの情報をMUXが処理せず，ドライバ内で処理する）という違いがある．NPTはおもに仮想ドライバやEthernet以外のリンク・レイヤのドライバ開発に向く．

注2：ENDはWind Riverオリジナルのドライバ・モデルで，BSDのドライバ・モデルとの類似点は多いが，ドライバ・レベルでのポール・モードをサポートするなど，優位点が多い．VxWORKSはEND，NPT，BSDの三つのドライバ・モデルをサポートする．

注3：BSPはターゲットとなるSingle Board ComputerやEmbedded Computerのハードウェアを直接制御するモジュールの集合体．

注4：Tornado Project FacilityとはVxWORKSのコンフィグレーションを自動化するGUIベースのツールで，コンポーネントの依存性の自動解析，サイズ計算，オート・スケーリングを行う．

表1　必須ネットワーク・コンポーネントの一覧と初期化シーケンス

初期化順	コンポーネント名	モジュール名	Configlette[注6]	関数名	コンポーネントの説明
1	INCLUDE_BOOT_LINE_INIT	N/A	usrBootLine.c	usrBootLineParse	ブート・パラメータ取得
2	INCLUDE_NET_SETUP	N/A	usrNetLib.c	usrNetLibInit	ネットワーク・バッファとnetBufLibの初期化
3	INCLUDE_BSD_SOCKET	bsdSockLib, sockLib	usrBsdSocket.c	usrBsdSockLibInit	BSDソケット
4	INCLUDE_HOST_TBL	hostLib	N/A	hostTblInit	ホスト名関連処理
5	INCLUDE_IP	ipLib	usrNetIpLib.c	usrIpLibInit	IPv4
6	INCLUDE_UDP	udpLib	usrUdp.c	udpLibInit	UDP
7	INCLUDE_TCP	tcpLib	usrTcp.c	tcpLibInit	TCP
8	INCLUDE_ICMP	icmpLib	usrIcmp.c	icmpLibInit	ICMP
9	INCLUDE_NET_LIB	netLib	N/A	netLibInit	tNetTask，splなどの初期化 リスト1参照
10	INCLUDE_MUX	muxLib, muxTkLib	N/A	muxLibInit	MUX
11	INCLUDE_END	endLib	usrEndLib.c	usrEndLibInit	END
12	INCLUDE_NET_INIT	N/A	usrNetBoot.c	userNetBoot	ブート方法指定
13	INCLUDE_DHCPC_LEASE_CLEAN	N/A	usrNetBootUtil.c, usrNetBoot.c	usrDhcpcLeaseClean	DHCP未使用時の処理
14	INCLUDE_NETDEV_NAMEGET	N/A	usrNetBoot.c	usrNetDevNameGet	ブート・パラメータからhost名設定
15	INCLUDE_NETMASK_GET	N/A	usrNetBootUtil.c, usrNetBoot.c	usrNetmaskGet	ブート・パラメータからnetmaskを抽出
16	INCLUDE_END_BOOT	N/A	usrNetBoot.c, usrNetEndBoot.c	usrNetEndDevStart	ENDによるブート
17	INCLUDE_LOOPBACK	If_loop	usrNetLoopbackStart.c	usrNetLoopbackStart	ループバック・インターフェース
18	INCLUDE_NETDEV_CONFIG	N/A	usrNetConfigIf.c	usrNetConfig	ブート・パラメータからIPアドレスなどを設定

表1に，一般的なネットワーク・コンフィグレーションで必要となるコンポーネント[注5]の一覧を示します．コマンド・ライン・ビルドでは，BSPディレクトリ内のconfig.hにこれらのコンポーネントをマクロとして定義してください(例：#define INCLUDE_IP)．Tornado Project Facilityでは，従来のコンフィグレーション・ファイルconfig.All.h，config.h，userNetwork.cとインクルード依存ルールuserDepend.cによって従来のコマンド・ライン・ビルドよりも強力なモジュール相互の依存性の強力なチェックとその解決，ビルド・システムのサイズの最適化を行います．

コマンド・ライン・ビルドで，もっとも簡単なネットワーク・コンフィグレーションの方法はINCLUDE_NETWORKを定義することでしたが，Tornade Project FacilityにおいてINCLUDE_NETWORKはダミーのコンポーネントで，ネットワークに必須のコンポーネントをインクルードするだけです．ほかの必須コンポーネントは，芋づる式にインクルードされます．詳しくは，Tornade Project FacilityのComponent Description File 00vxWorks.cdf，00network.cdf，01network.cdfおよび生成されたプロジェクト・ディレクトリにあるprojComp.hとprjConfig.cを参照してください．

リスト1で示したように，プロトコルのコアの部分の初期化はINCLUDE_NETLIBで行われます．プロトコルの初期化関数の大半はBSDに由来するもので，*Stevens TCP/IP Illustrated Vol.2*を参考にすると，一層の理解が得られます．

● tNetTask —— VxWORKS TCP/IPプロトコル・スタック・タスク

さて次は，VxWORKSのTCP/IPプロトコル・スタックをリスト2と図2を参照しながら見ていきましょう．VxWORKSのTCP/IPプロトコル・スタックは，tNetTaskという1タスクとして実装されています．tNetTaskは，ネットワークISR

注5：Component(コンポーネント)とはVxWORKSの機能の単位である．コンポーネントはTornado Project Facilityによって，従来のコマンド・ライン・ビルドではできなかった柔軟なシステム構築をユーザに提供するための重要な要素である．たとえばINCLUDE_FTPはFTPクライアント機能を提供するコンポーネントである．コンポーネントはVxWORKSへのインクルード(ビルド時にリンク)・エクスクルードが可能で，ユーザ設定可能なパラメータをもちうる．

注6：Configlette(コンフィグレット)とはProject Facilityにおいて，コンポーネントの定義ファイルComponent Description File(CDF)から参照されるコンポーネント初期化モジュール．コンポーネントに属するモジュールがもつ初期化ルーチン以外に初期化処理が必要な場合，Configletteによって対応する．

によってキューされた受信パケット処理のJOBをデキューしてパケットをmbuf注7化するところからソケット・レイヤでの処理までを担当します．

このメカニズムは，BSDやLinuxと大きく違うところです(Windowsがどうやっているのかは不明)．

これに対して，送信パスでは送信要求を行ったアプリケーション・タスク内でネットワーク・インターフェースにパケットを渡すまでを担当します．このため，送信アプリケーション・タスクのタスク・プライオリティをtNetTaskの50よりも優先順位を上げると若干のスループット向上を確認できるかもしれませんが，原則としてネットワーク・ジョブを集中処理しているtNetTaskが，ほかのネットワーク・アプリケーション・タスクより高い優先順位となるように設定してください．WDBタスクがtNetTaskよりも高い優先順位なのはネットワーク・アプリケーションやtNetTaskをデバッグ可能とするためのものです．

前項のBSD TCP/IPプロトコル・スタックとの比較のところで説明したとおり，VxWORKSのISRでは最小限のこと，つまりtNetTaskのリング・バッファ(ネットワーク・ジョブ・キュー)に受信処理要求をキューして直ちにISRを終了します．これは組み込みRTOSのISRは極力短期間のうちに仕事を終わらせるべきである，という考えかたからきています．ちなみに，ネットワーク・ジョブ・キューのリング・バッファ・サイズはTornado2.2のデフォルトで85固定です．このリング・サイズ

リスト 1　netLibInitによるtNetTaskの初期化

```
<target/src/netwrs/netLib.c>
/*************************************************************
*
* netLibInit - TCP/IP プロトコル・スタックの初期化
*
* 当ルーチンでTCP/IPプロトコル・スタックの初期化，tNetTaskのJob
* キュー作成，tNetTask起動を行う．
*/

STATUS netLibInit (void)
    {
    /*
     * このルーチンで呼び出される初期化ルーチンの大半はBSDにも存在する
     * ので，それらはStevens TCP/IP Illustrated Vol.2参照のこと．
     *
     * VxWORKSユニークなのが:
     * splSemInitはVxWORKS版SPL(実体はMUTEXセマフォ)の初期化．
     * addDomainはBSD4.4におけるADDDOMAINマクロに等しい．
     * netTypeInitはnetwork typeリストの初期化(現状このリストは
     * PROXY ARP以外で使われていない)
     *  mcastHashInitはマルチキャスト関連のハッシュ・テーブル初期化．
     */

    splSemInit ();
    mbinit ();
    ifinit ();
    addDomain (&inetdomain);
    domaininit ();
    route_init ();
    netTypeInit ();
    mcastHashInit ();

   tNetTaskのJOBキュー初期化(実体はリング・バッファ)
    REBOOTフックにifresetImmediate関数をフックする
   tNetTaskをtaskSpawnする

    }
```

リスト 2　tNetTaskの仮想コード

```
void netTask (void)
    {

    FOREVER
        {
        ISRがJobをキューするまで待つ．

        netTaskSemIdというセマフォをWAITFOREVER で待ち，
        ISRがnetJobAddでJobをキューする際にnetJobAddが
        netTaskSemIdをギブする，というシンプルなロジック．

        while (Jobキューが空になるまで繰り返し)
            {
            Jobキュー( netJobRing )にスタックされた関数ポインタ
            (ほとんどの場合，ドライバのタスク・レベル受信ルーチン)
            で関数コール．　Socketレイヤにパケットを引き継いだら，
            呼んだ関数がここに返ってくる．この間にもISRがJobを
            キューしてくる．
             }
        }
    }
```

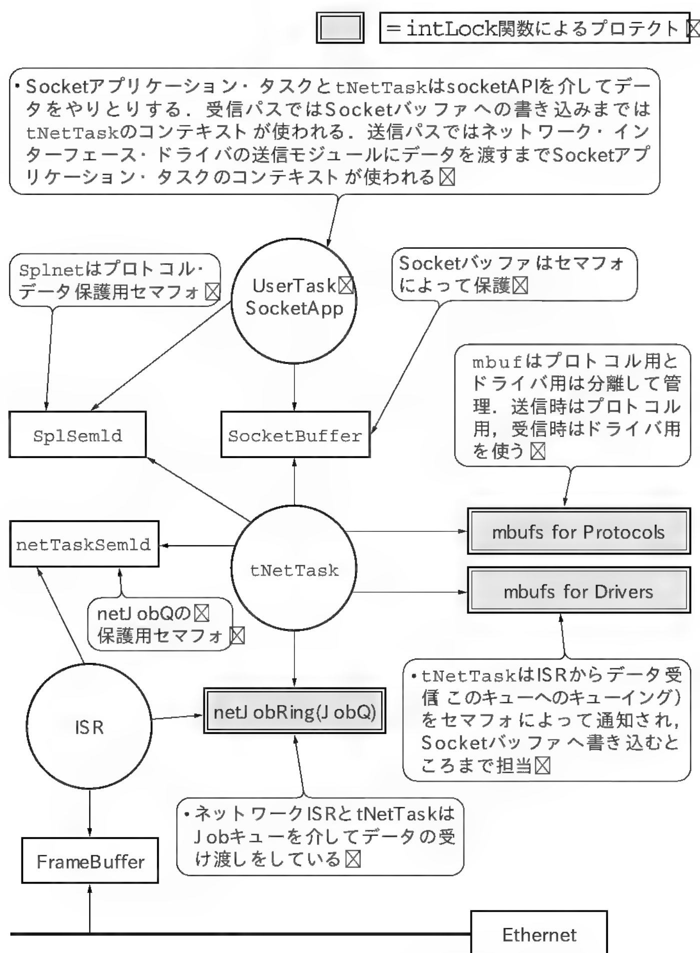


図2　VxWORKSネットワーキングのタスク・データ・フロー

注7：mbufとはBSDのIPC用メモリ管理モジュールとその構造体を指す．BSDにおけるネットワーク・パケットは，mbufによってハンドリングされる．VxWORKSのネットワーク・バッファ管理モジュールnetBufLibでは，mBlkというmbufに似たコンセプトのデータ構造体をもっている．

が意味するのは，tNetTaskがデキューしない状態で最大85パケットの受信要求が処理可能ということです．

tNetTaskは，タスク・プライオリティ50でつねにネットワーク・ジョブ・キューから受信処理要求をデキューしているので，85というサイズで問題ないはずです．ただし，今後ギガビットEthernetなど，より頻繁に割り込み要求が発生するシステムにおいては，上位のアプリケーションがSocketバッファを読むよりも早くキューを使い果たす可能性があります．

ちなみにWind Riverの最新TCP/IPプロトコル・スタックWindNet IPv6では，リング・バッファ・サイズが可変となり，TornadoのTarget Configuration ToolであるTornade Project FacilityにてVxWORKSをビルドする際に設定可能です．

今後，VxWORKS5.xに標準装備されるTCP/IPプロトコル・スタックにこの変更が反映されると期待しています．

MUXとEND，NPTドライバ・モデル

● MUXアーキテクチャとEND/NPTドライバ

MUXはWind River固有のアーキテクチャで，図3にあるように，データ・リンク層とネットワーク層の中間に位置し，ネットワーク層とデータ・リンク層が互いに依存せずにデータ交換をすることでシステムのプロトコル構成の自由度を高めています．

END/NPTという二つのドライバ・モデルとMUXによる標準インターフェースを定義することで，VxWORKS TCP/IPプロトコル・スタックにユーザ作成のプロトコルやドライバをインテグレートすることが容易になりました．

ただし，MUXとEND/NPTのドライバ・モデルであっても，BSDから継承されているIPとARPの相互依存は完全には解消できていません．具体的にいうと，ifnet構造体へのポインタ*ifpをIPとIPサブレイヤipProtoでarpcom構造体にキャストして参照しています．

BSDベースのTCP/IPプロトコル・スタック実装だと，この2層間で双方の内部データ構造に依存しています．RFCで規定されているプロトコル自体にはデータ・リンク層への依存がないので，ユーザも使い始めてからこの実装上の制限に気が付くことが多いようです．

Ethernet LAN上で稼動することを想定した組み込みシステムなら良いのですが，組み込みネットワーク機器の用途はIPやEthernetに限定されているわけではありません．

図4は，END_OBJを中心に関連するデータ構造体の関係を示したものです．この図で，END_OBJをはじめ，MUX/ENDアーキテクチャで使用されているデータがオブジェクト指向であるのがわかります．ハードウェア依存のデータはDEV_OBJ

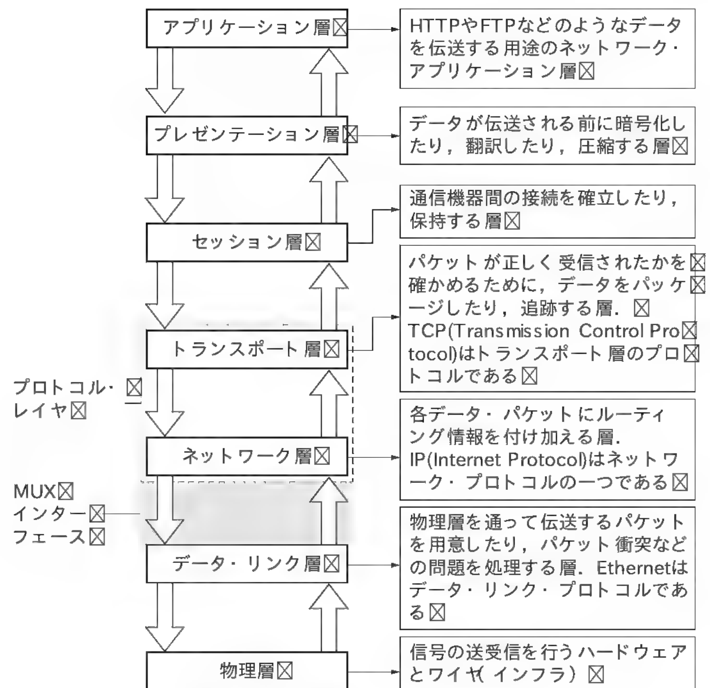


図3　MUXのOSI7層における位置づけ

として管理し，muxBindでMUXにバインドされた上位プロトコル・レイヤのサービス関数はNET_PROTOCOLとして管理し，逆にMUX経由で上位プロトコルから呼び出されるENDのサービス関数はNET_FUNCSとして管理しており，muxDevLoadによってENDのloadルーチンが呼ばれた際にEND_OBJに登録されます(通常はEND_OBJ_INITマクロ経由で行われる)．

ドライバ・モデルとしてのENDとNPTの違いは，ENDがデータ・リンク層のヘッダ処理がMUX内で行われるのに対して，NPTではデータ・リンク層のヘッダ処理がNPTドライバ内で行われることです．ユーザがEthernetのドライバを書くときには，ENDモデルをベースにすることをお勧めします．

ENDモデルではテンプレートとなるtemplateEnd.cと，すでにVxWORKSで実装済みのEthernetドライバのソース・コードが同梱されており，これを参考に短期間でドライバ開発が行えます．逆に仮想ドライバなどは，NPTのほうが作りやすいと思われます．

VxWORKSにできることは，MUX/ENDのバインディングだけではありません．図5が示すようにSocketのバック・エンド[注8]とプロトコルもユーザ自身が実装して，Socket APIからもMUXからも呼び出すことが可能です．

これにより，Wind Riverの膨大なBSPとネットワーク・インターフェース・ドライバ，Socketベースのアプリケーション

注8：ソフトウェアが2階層以上で構成されているケースで，ユーザもしくはユーザ・アプリケーションとのインターフェースをフロント・エンドといい，対してフロント・エンドよりも後方に位置するものをバック・エンドと称する．たとえば，低水準入出力関数read/writeがフロント・エンドで，その下で動くハードウェアごとの実際のI/O処理がバック・エンドともいえる．

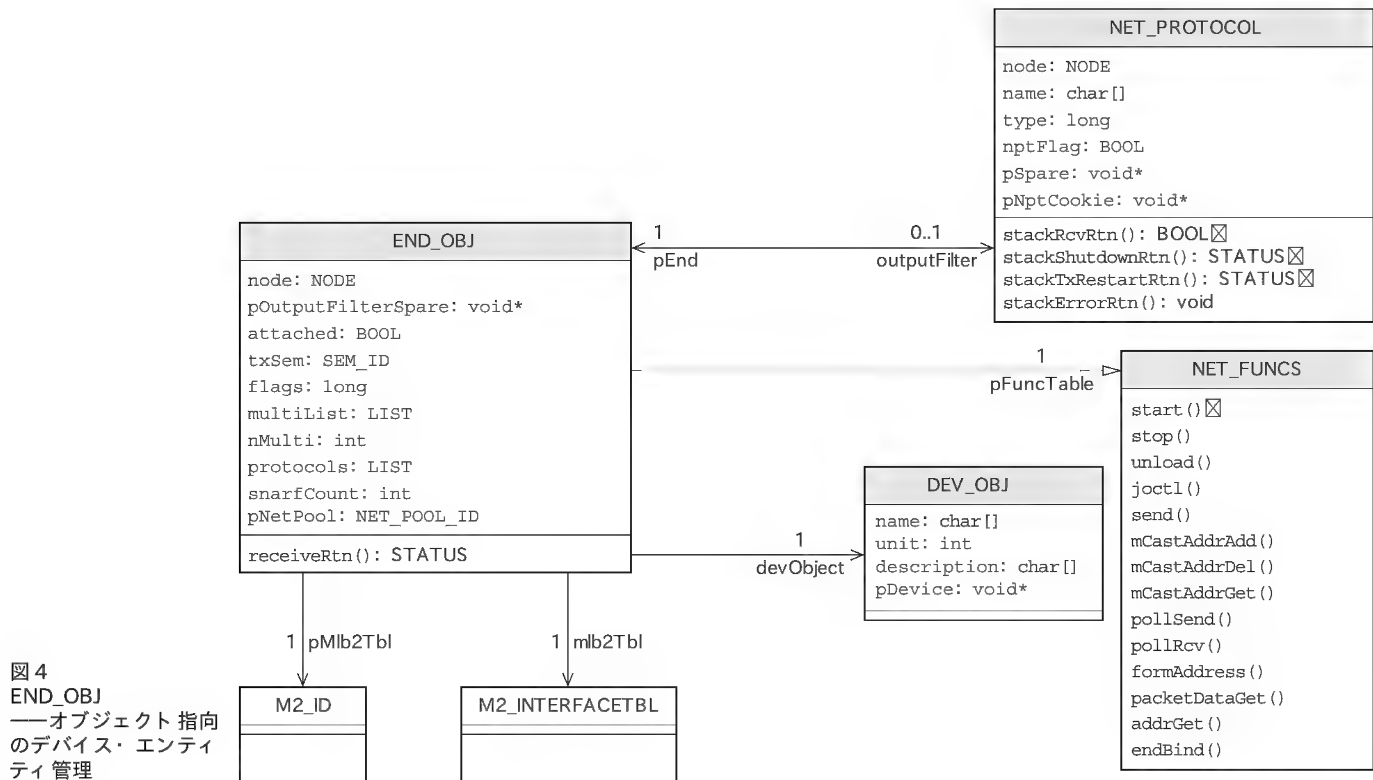


図4 END_OBJ ――オブジェクト指向のデバイス・エンティティ管理

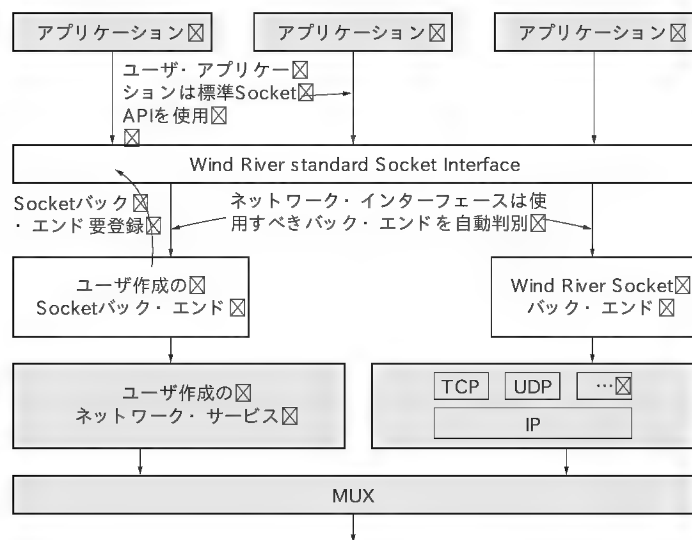


図5 VxWORKS Socket-MUX間のモジュール構成

を修正することなしに，ユーザのプロトコルやSocketバック・エンドとのインテグレーションが可能です．Wind RiverはSocket APIへのバック・エンドとして，BSD Socketバック・エンド(INCLUDE_BSD_SOCKET)を提供しています．

● ipAttach関数内の処理

おそらくMUX/ENDに関連して，いちばんわかりにくいのがipAttach関数でしょう．ここでは，リスト3をベースにipAttach関数内の処理について説明します．図4，図5と照らし合わせて見ていくと，より理解が深まると思います．詳細はリスト3を参照するとして，ここではipAttach関数の中で以下の主要な処理を解説します．

- ▶ ipDrvCtrlテーブルへのエントリ追加
- ▶ muxBind/muxTkBind関数によるIP(ARP)-MUX-ENDのバインディング(関連付け)
- ▶ 該当END_OBJのifnet構造体の初期化
- ▶ if_attach関数の呼び出し

マニュアルでは見えにくい部分ですが，ユーザがMUXにバインド可能なプロトコルを開発する際には，同様のアタッチ関数が必要となります．

まず，ipDrvCtrlテーブルへのエントリ追加ですが，IPレイヤではIPにバインドされているEND_OBJをIP_DRV_CTRL構造体で管理しています．IP_DRV_CTRLはipProto.hに定義されていますが，基本的にInterface Data Record(IDR)とMUX/ENDやNPT固有の情報を合わせもったものです．

IDRは現状struct arpcomで，ご存じのとおりstruct ifnetを包含しています．BSDと同様，インターフェースに付随するサービスはIDR(struct ifnetにキャスト)に登録された関数群で行います(例: ifp->if_ioctl)．

バインドされたEND_OBJ間はお互いを関知しないし，いったんTCP/IPプロトコルの内部に入ってしまうとMUX/END固有の情報を関知しません．そのため，IPサブレイヤがこの

リスト 3　ipAttach ―― MUX-IP レイヤのバインド・メカニズム(仮想コードなので，コンパイルは通らない)

```
<target/src/netwrs/ipProto.c>
/***************************************************************
*
* ipAttach - IPをMUXにアタッチし，IP_DRV_CTRLと ifnetを初期化する.
*/

int ipAttach
    (
    int unit,           /* ユニット番号  */
    char *pDevice       /* デバイス名 */
    )
    {

    endFindByNameで該当するEND_OBJを取得.

    ipDrvCtrlテーブルを走査，すでにアタッチ済みかチェックし，
    新規なら空きスロットを取得し，当END_OBJ用に使用する.

    /*
    * ipDrvCtrl[]はIPにバインドされたEND_OBJを管理するテーブル.
    * コンポーネント・パラメータ ipMaxUnitsで ipDrvCtrlのエントリ数を
    * ユーザ設定可能.
    */

    /*
    * IPレイヤをMUXにバインド. NPTの場合，リンク・レイヤのヘッダは
    * パケットがMUXに渡される前に除去されている.
    */

    if (ドライバがNPTスタイルの場合)
        {
         muxTkBind関数で IPを MUXにバインド. バインドされる関数は以下の通り
         ipTkReceiveRtn
         ipTkShutdownRtn
         ipTkTxRestart
         ipTkError
         }
    else     /* ドライバがENDスタイルの場合 */
         {
         muxBind関数で IPを MUXにバインド. バインドされる関数は以下の通り.
         ipReceiveRtn
         ipShutdownRtn
         ipTxRestart
         ipError

         ENDの場合は送信用にリンク・レベルのヘッダ情報を保持する必要あり.
         pDrvCtrl->pSrc, pDrvCtrl->pDstに mbufを dataプールから
         アロケート.
         }

    pDrvCtrl->pDstAddrにMAX_ADDRLENと同じかそれより大きいクラスタを
    dataプールからアロケート.

    muxIoctl関数でENDのデバイス情報とMIBデータを取得.
    ENDがRFC2233をサポートしていればEIOCGMIB2233を， RFC1234を
   サポートしていれば，EIOCGMIB2を使う. MIBデータはmib2Tblに格納する.

    /*
    * idrをstruct ifnetへのポインタにキャスト，BSDにおけるドライバの
    * attach関数で行っているifnetの初期化をここで行う.
    * idrはInterface Data Recordの略で，struct arpcomに相当する.
    */

    pIfp = (IFNET *)&pDrvCtrl->idr;

    pIfp->if_unit    = unit;
    pIfp->if_name    = pName;
    pIfp->if_mtu     = mib2Tbl.ifMtu;
    pIfp->if_baudrate = mib2Tbl.ifSpeed;
    pIfp->if_init    = NULL;
    pIfp->if_ioctl   = (FUNCPTR) ipIoctl;
    pIfp->if_output  = ipOutput;
    pIfp->if_reset   = NULL;
    pIfp->if_start = (FUNCPTR)ipTxStartup;

    /*
    * muxAddrResFuncGetを ETHERTYPE_IP(0x800)を指定して呼んでいるが，
    * ipAttachはNPTも使うため，ETHERNETと限定する必要はない.
    */

    pIfp->if_resolve = muxAddrResFuncGet (mib2Tbl.ifType,
                                                      ETHERTYPE_IP);
    pIfp->if_flags  = flags;
    pIfp->if_type = mib2Tbl.ifType;
    pIfp->if_addrlen = mib2Tbl.ifPhysAddress.addrLength;

    /* pCookieはIP_DRV_CTRLへのバック・ポインタとしてセットされる */

    pIfp->pCookie = (void *)pDrvCtrl;

    リンク・アドレスをmib2Tblにコピーする.

    if (ドライバがNPTスタイルの場合)
        {
         muxTkBind関数で ARPを MUXにバインド. バインドされる関数は以下の通り
         ipTkReceiveRtn
         arpTkShutdownRtn
         ipTkTxRestart
         NULL(エラー関数エントリなし)
         }
    else     /* ドライバがENDスタイルの場合 */
         {
         muxBind関数で IPを MUXにバインド. バインドされる関数は以下の通り.
         ipReceiveRtn
         arpShutdownRtn
         ipTxRestart
         ipReceiveRtn
         NULL(エラー関数エントリなし)
         }

    muxIoctl EIOCGHDRLENでヘッダ長を取得し，pIfp->if_hdrlenに設定.

    /*
    * インターフェースにアタッチ
    */

    if (if_attach (pIfp) == ERROR)
        goto ipAttachError;

    アタッチするインターフェースのアドレス・リストを走査，
    リンク・レベル・アドレスをmib2Tblにコピー.

    pDrvCtrl->attached = TRUE;  /* アタッチ処理正常終了をフラグ ON */
    return (OK);
    }
```

テーブルでEND_OBJを管理する必要があるのです．

つぎに，muxBind/muxTkBind関数でMUXとインターフェースするIPサブレイヤの関数群をMUXにバインドします．これでIP-MUX-ENDと関連付けされ，送受信データ・パスが完成します．

そして，該当するEND_OBJのIDR初期化です．この処理を見て，BSDを良く知るユーザはピンときたと思います．そうです．ipAttach関数はBSDドライバのアタッチ関数の処理を包含しているのです．struct ifnetのif_で始まる関数ポインタの実体がわからず，困っていたVxWORKSユーザも少なくなかったと思うので，今回はコードをそのまま公開することにしました．バージョンの違いにより若干差異はあると思いますが，どのバージョンのipAttach関数でもIDRの初期化は，リスト3に示した処理とほぼ同じです．

最後にif_attach関数を呼びます．これもBSDドライバのアタッチ関数と同じです．

リスト 4　VxWORKS TCP/IP プロトコル・スタック受信パス（仮想コードなので，コンパイルは通らない）

```
・関数コール・シーケンス

(ISR)
   templateInt

(tNetTask)
   templateHandleRcvInt
      muxReceive
         ipReceiveRtn
            do_protocol_with_type
                ipintr/arpintr


<src/drv/end/templateEnd.c>
/******************************************************************
*
* templateInt - ネットワーク割り込みハンドラ
*/

LOCAL void templateInt(略)
    {
    コントローラのステータスをチェック，
    無効な割り込みであれば，リターン.

    受信割り込みだったら，netJobAdd関数を使って，tNetTaskの
    Jobキューに templateHandleRcvInt関数ポインタをキューイングする.

    送信割り込みを処理.
    }


/******************************************************************
*
* templateHandleRcvInt - タスク・レベルでの受信ルーチン
*
* このルーチンはネットワーク ISR によって tNetTask のジョブキューにキュー
* され，tNetTask タスクのコンテキスト内で呼ばれ，ネットワーク ISR が受信
* したパケットを処理する.
*/

LOCAL void templateHandleRcvInt(略)
    {

    ドライバのバッファ・プールを使って，受信パケットを mbuf 化
    END_RCV_RTN_CALL で MUX にパケットを渡す．後続の受信パケッ
    トをすべて処理するまで繰り返し.

    }


<src/netwrs/muxLib.c>
/******************************************************************
*
* muxReceive - ドライバから渡されたパケットを処理，バインドされた
*              プロトコルに渡す．ドライバは END_RCV_RTN_CALL 経由で
*              このルーチンを呼ぶ.
*/

STATUS muxReceive(略)
    {

    for (プロトコル・リスト NET_PROTOCOL を走査)
        {
        MUX にバインドされた各プロトコル・リストの受信ルーチン stackRcvRtn
        にパケットを渡す.

        返値が TRUE ならパケットは該当プロトコルによって処理されたと解釈し，
        そうでなければ FALSE が返され，次のプロトコルの受信ルーチンに
        パケットが渡る.

        プロトコル・リスト登録順には以下のルールがある.
        1. SNARF
        2. その他
        3. PROMISC
        }
    }


<src/netwrs/ipProto.c>
/******************************************************************
*
* ipReceiveRtn - MUX→IP へのパケット渡し
*                データ・リンク層のヘッダを除去し，IP ヘッダが mbuf 先頭にくる
*                ように調整後，do_protocol_with_type 関数を呼ぶ.
*/

BOOL ipReceiveRtn(略)
    {

    データ・リンク層のヘッダ全体が先頭の mbuf になければ，m_pullup 関数で
    ヘッダ全体を mbuf チェイン先頭にもってくる.

    データ・リンク層ヘッダを取り除く.

    do_protocol_with_type 関数を呼び，その中で IP か ARP に分岐する.
    }


<src/netinet/if_subr.c>
/******************************************************************
*
* do_protocol_with_type - 受信パケットを IP か ARP に渡す.
*/

void do_protocol_with_type(略)
    {

    IP なら ipintr 関数を呼び，ARP なら arpintr 関数を呼ぶ.
    IP より上位のパケット処理については Stevens TCP/IP Illustrated Vol.2
    を参考に BSD のソース・コードにて理解を深めていただきたい.

    }
```

VxWORKS TCP/IP プロトコル・スタックの受信パス

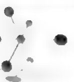
リスト 4の仮想コードを見ながら VxWORKS TCP/IP プロトコル・スタックの受信パスを見ていきましょう．

ドライバの ISR（templateEnd.c では関数 templateInt）は，tNetTask の Job キューにタスク・レベルでの受信ルーチンの関数ポインタ templateHandleRecvInt をキューして，直ちに終了します．

tNetTask は Job キューからその関数ポインタを取り出して，tNetTask のタスク・コンテキスト内で関数コールをします．

Wind River から提供されている複数のドライバを比較すればわかりますが，タスク・レベル受信ルーチンの中身はハードウェアに依存するデータ・リンク層フレーム取得の処理と，取得したフレームを mbuf 化する処理で構成されます．mbuf 化に使うバッファは，TCP/IP プロトコル・スタックとは別にドライバが確保したものです．

ドライバのバッファをプロトコルとは別に管理するのは，プロトコルでの mbuf 使用率に影響を受けずに，tNetTask の Job キューの数だけつねに受信フレーム用の mbuf を確保できることを保証し，フレームの取りこぼしを防止するためです．



また，ドライバで必要とするmbufサイズは受信フレーム・ディスクリプタと同じでなければならず，PHY[注9]によってサイズが異なります．

これは，netBufLibで定義されているクラスタ・サイズにはフィットせず，結果として2048バイトのクラスタ[注10]を使うこととなり，メモリのむだとなります．また，netBufLibのmBlkClGet関数内では要求されたクラスタ・サイズを満たすクラスタを探すのに，64，128，256，512，1024，2048バイトと最悪ケースで6回の判定が発生しますが，単一のクラスタ・サイズでクラスタ・プールをもてば，判定は1回で済みます．

mbuf化されたパケットは，END_RCV_CALLマクロでマッピングされているパケット・ディマルチプレクサ[注11]（ENDではmuxReceive関数，NPTではmuxTkReceive関数）に渡されます．そこで，ここではENDのケースを見てみましょう．まずmuxReceiveではデータ・リンク層のヘッダを取り除き，次に受信したネットワーク・インターフェースのEND_OBJに登録されたプロトコル・リストを走査し，SNARF，Promicuousまたはフレーム・タイプが一致するプロトコルの関数ポインタstakckRcvRtnで関数コールを行い，その関数にパケットを渡します．このstakckRcvRtnはmuxBind関数で登録されたプロトコルの受信ルーチン（MUXからパケットを受信するルーチン）で，自分でパケットを処理するのであれば，パケット処理後にTRUEをmuxReceive関数に返します．FALSEであった場合はmuxReceive関数は次のプロトコル（該当するプロトコルはSNARF，Promicuous，フレーム・タイプが一致するものだけ）の受信ルーチンを呼び，パケットを渡します．受信ルーチン内で何をするかはプロトコルに任されており，厳密な規定はありません．

もし，パケットを処理すべきプロトコルが登録されていなければ，パケットを破棄します（そのパケットが入ったmbufを解放して，ドライバのクラスタ・プールに戻す）．

では，VxWORKS TCP/IPプロトコル・スタックでは，どう受信パケットが処理されるのかを見ていきましょう．

ipProtoはIPサブレイヤとしてMUXとのインターフェースを担当します．ipAttach関数はIPサブレイヤ初期化の一環としてmuxBind関数を呼んでIPサブレイヤ関数をMUXにバインドします．MUX-IP間のデータ交換は，バインドされたルーチンを経由して行われます．

Tornado2.2以前のVxWORKS TCP/IPプロトコル・スタックでは，ipRecieveRtn関数はMUX-IP間の受信パスのデータ交換を担当し，渡されたmbufがIPヘッダ先頭を指すように調整し，do_protocol_with_type関数を呼びます．do_protocol_with_type関数は，フレーム・タイプでIP（ipintr関数）かARP（arpintr関数）を呼び出しています．

鋭い読者はここでわかると思いますが，ARPは関数ポインタarpReceiveRtnをMUXにバインドしているので，関数ポインタipReceiveRtnのパスでARPへ分岐することはありません．WindNet IPv6ではBSDスタイルのドライバはサポートしていないので，END/NPTの受信パスのみを考慮すればよいため，関数ポインタipRecieveRtnから直接IPの受信ルーチンを呼んでおり，do_protocol_with_type関数そのものを廃止しています．対して，Tornado2.2ではBSDスタイルのドライバをサポートしているため，BSDスタイルのタスク・レベル受信ルーチンでdo_protocol_with_type関数を呼び，この関数の中でIP/ARPの分岐を行っています．

つまり，ENDにおけるmuxReceive関数とipReceiveRtn関数は，BSDスタイル・ドライバのタスク・レベル受信処理を分割したものです．

Tornado2.2以前のVxWORKS TCP/IPプロトコル・スタックでは，do_protocol_type関数内で再度IF_ENQUEUEマクロでパケットをipintrqかarpintrqにキューしていました．しかし，Ethernetコントローラのリング・バッファからtNetTaskへのキューイングはネットワークISRがtNetTaskのJobキューになされた時点で終了しており，またネットワークISRからtNetTaskへのJobキューイングの通知は高速なセマフォで行われているため，BSDのようにipintr関数をソフトウェア割り込みで呼ぶ必要がありません．VxWORKSでもipintr関数は存在しますが，ソフトウェアISRではなく，tNetTaskのタスク・コンテキストで呼ばれる関数です．

IPの受信ルーチン（Tornado2.2ではipintr関数，WindNet IPv6ではip_input関数）が呼ばれた時点からパケットがSocketレイヤに到達するまでの処理はあまりBSDと変わりません．もちろん，オリジナルの4.4BSDのコードと現状のVxWORKSのTCP/IPプロトコル・スタックのコードはまったく同じということではありません．しかし，IPからSocketレイヤまでのアーキテクチャはほぼ同じと思ってもかまいません．

したがって，読者の皆さんには名著*TCP Illustrated Vol.2*と*The Design and Implemention of the 4.4BSD Operating System*を参考にしながら，VxWORKS TCP/IPプロトコル・スタックのベースとなったBSD4.4の理解をソース・コード・レベルで行っていただき，IP-Socketレイヤの詳細についてはここでは割愛します．

注9：PHYはPhysical Layer（物理層）の略で，OSI 7層のうち，最下層に位置する．

注10：ここでいうCluster（クラスタ）とはnetBufLibのデータ・バッファを指す．BSDと異なり，VxWORKSのmBlkはデータ領域がなく，データ・バッファへのポインタを保持する．このデータ領域をクラスタと呼ぶ．

注11：マルチプレクサ（Multiplexer）は複数の信号を1本のライン（メディア）で送信することで，デマルチプレクサ（Demultiplexer）は1本のラインで送信された複数の信号を分岐させること．パケット・デマルチプレクサとは単一のライン（この場合はEthernet）で送信されたパケットを上位プロトコルに渡す処理を行うモジュールのことで，VxWORKSの場合，MUXを指す．

リスト 5　VxWORKS TCP/IP プロトコル・スタック送信パス(仮想コードなので，コンパイルは通らない)

```
(User's socket app)
   send - ip_output までは BSD とほぼ同じ.
      ip_output
         ipOutput - ifp->if_output 関数ポインタ経由で呼び出し
            ipTxStartup - ifp->if_start 関数ポインタ経由で呼び出し
            muxSend/muxTkSend
               END send - pEnd->pFuncTable->send 関数ポインタ経由で
                  ドライバのフレーム送信関数を呼び出し


<target/src/netwrs/ipProto.c>
/*****************************************************************
*
* ipOutput - IP データ・グラムを END/NPT 経由で送信
*
* END ではリンク・レベルのフレームを形成してからドライバにデータを渡す.
* NPT では送り先リンク・レベル・アドレスを mBlk チェイン先頭に追加する.
* このルーチンでは ifp は実際には arpcom として扱われる.
* BSD の ether_output 関数と比較しながら理解すべし.
*/

int ipOutput
    (
    (略)
    )
    {

    ifp->if_lastchange = tickGet();

    ip_output から渡されたルート( NEXT HOP)が有効か，Gateway かチェック.

    送り先 IP アドレスの SA_FAMILY によって処理が分岐( Switch-Case).

    AF_INET のケース:
      if_resolve ポインタ経由で Address Resolution を行う.
      END の場合，ipAttach 関数が muxAddrResFuncGet 関数の返値( 関数ポイン
      タ) をセットしている.  通常 arpresolve 関数が if_resolve 経由で呼ばれ
      る.

      もしインターフェースが Simplex なら，送信パケットのコピーをループ
     バック・インターフェースに送る.


    AF_UNSPEC のケース:
      送信パケットがリンク・レベルのヘッダをすでに作成済み.

    未サポートのケースでは EAFNOSUPPORT のエラーを返す.


    if( NPT の場合)
        {
        送り先アドレスとプロトコル・タイプを送信パケットに書き込む.
        }
    else /* END の場合 */
        {
        muxAddressForm を使って，完全なイーサネット・フレームを作成.
        }

    s = splimp();

    IF_ENQUEUE マクロでインターフェースに送信パケットをキューイング.
    ifp->if_start 関数ポインタ経由で muxSend を呼び，送信パケットを
                                                        ドライバに渡す.
    splx(s);

    }


/*****************************************************************
*
* ipTxStartup - パケット送信開始
*
* 当関数が呼ばれるのは複数ケースある.
*
* 1) ユーザ・タスクが送信するケース
*     muxSend 関数で現在キューされているパケットをすべて送信するか,
*     リソース不足に直面するまで送信を試みる.
*
* 2) ICMP reply のように，tNetTask コンテキスト内で送信が発生するケース
*     処理事態は上記 1 と変わらないが，タスク・コンテキストが tNetTask で
*     あることに注意.
*
* 3) デバイス・ビジーなどにより，再送信時に ipTxRestart( END)か
*     ipTkTxRestart( NPT)から呼ばれる.
*/

void ipTxStartup
    (
    (略)
    )
    {

    s = splnet();

    /*
     * パケットが送信キューからなくなるか，リソース不足に直面するまで，繰り
     * 返し送信を試みる.
     */

    while (pIf->if_snd.ifq_head) /* 送信キューにパケットがあるか? */
        {

        パケットを送信キューから取り出す.

        if (NPT の場合)
            {
            NPT の場合は，パケット・タイプ( 必要であれば，リンク・レベル送り先
            アドレスも) を mBlkHdr.reserved から取り出し，一時データとし
            てコピー，後述の muxTkSend の引き数や，END_ERR_BLOCK で再送信の
            際にドライバに変更されたヘッダを修復するために使う.

            if (送り先アドレスがある場合)
                {
                送り先アドレスを mBlkHdr.mData からコピー，muxTkSend 関
                数への引数として渡す.

                m_adj 関数にて ipOutput 関数で追加したヘッダをスキップして
                元のヘッダを指す様にポインタとデータ長を調整.

                送り先アドレスに新しい mBlk を使っていた場合，元のヘッダを
                格納する mBlk に M_PKTHDR のフラグを立てる.
                }

            muxTkSend 関数で送信.
            }

        else  /* END の場合 */
            {
            END の場合はこの時点でリンク・フレームが完成しており,
            それを muxSend 関数で送信するのみ.
            }

        if (END_ERR_BLOCK)  /* 何らかの理由でデバイスが送信できない場合 */
            {

            いったん，送信を試みたパケットをキューしなおす.

            if (NPT の場合)
                {

                /*
                * パケットはドライバによって muxTkSend 関数を呼ぶ直前の
                * 状態から変更されている( リンクレイヤのヘッダが mbuf
                * チェインの先頭にある) ので，直前の状態に修復する.
                */

                ドライバがパケットのヘッダ部分を変更しているので,
                保存しておいたデータでヘッダ修復. データ部分を指す
                ポインタも再度調整.

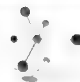
                }

                送信キューがフルなら，パケットを廃棄，そうでなければ
                IF_PREPEND で送信キュー先頭にパケットを追加.
            }

        else      /* 送信成功 */
            {

            NPT の場合，リンク・レベルの送り先アドレスの格納に使った mBlk をフリー.
            }

    splx(s);
    }
```



リスト 5　VxWORKS TCP/IP プロトコル・スタック送信パス(つづき)

```
/******************************************************************
*
* muxSend - パケットをネットワーク・インターフェースに送信
*
* Public APIにつき，リファレンス・マニュアル参照.
*/

STATUS muxSend
     (
     (略)
     )
     {

     OUTPUTフィルタがmuxBindでバインドされていたら，
     muxPacketDataGet関数で送信パケットからリンク・レベルのヘッダ情報を取
     得，OUTPUTフィルタを呼ぶ.

     /*
     * この時に使用される関数ポインタはstackRcvRtnで混乱を招きやすいが，こ
     * れは受信パスでmuxBindが使用する関数ポインタと共通のため.
     */

     ENDのsend関数を呼ぶ(END_OBJのpFuncTable->send関数ポインタ経由).
     }
```

VxWORKS TCP/IP プロトコル・スタックの送信パス

送信パスは，受信パスよりもシンプルです(リスト5)．Socketレイヤから IPまでの処理はBSDとあまり変わりません．

ip_output関数からif_outputの関数ポインタ経由でInterfaceレイヤのパケット送信ルーチンが呼ばれますが，これがBSDと異なっています．BSDのEthernetドライバではドライバのattach関数でether_output関数がif_output関数ポインタに登録されますが，VxWORKSではipAttach関数によってipOutput関数がif_output関数ポインタに登録され，MUXはこのルーチン経由でパケットをドライバへ渡し，BSDスタイルのドライバではBSD同様にether_output関数を呼びます．

ipOutput関数でやることはether_ouput関数とあまり変わりませんが，ether_output関数がarpresolve関数を直接呼ぶのに対し，ipOutput関数では関数ポインタif_resolve経由でAddress Resolution関数を呼び出します．もちろん，Address Resolutionが不要なデータ・リンク層のドライバについては，このif_resolveポインタはNULLです．したがってAddress Resolution処理は発生しません．MUXでは，このように細かいところで非Ethernet系のデータ・リンク層のプロトコルに配慮しています．

それでは，誰が関数ポインタif_resolveを初期化するのでしょうか．コマンド・ライン・ビルドではusrNetwork.cで，Project FacilityビルドではコンポーネントINCLUDE_ENDのConfigletteであるusrEndLib.cの中で，muxAddrResFuncAdd関数を呼びます．

この関数がEthernet用のAddress Resolution(ARP)関数ipEtheResolvRtnを登録します．登録はインターフェース・タイプごとに1回行うだけで十分です．インターフェースをUPするたびには必要ありませんが，MUXにアタッチする前に登録が必要です．さもないとattach関数内でmuxAddrResGet関数を呼ぶとNULLが返ってきます．

関数ポインタif_resolveにとって，NULLは「当インターフェース・タイプはAddress Resolution不要」という意味で，エラーではありません．したがって，attach関数もエラーか否かの判断がつかず，ランタイムでTCP/IPプロトコル・スタックがAddress Resolutionに失敗します．

ipEtherResolvRtn関数は，純粋にarpresolve関数のWrapper関数で，特別なことは何もしていません．引き数がarpresolve関数より一つ多いのはコール・バック関数を呼ぶためですが，現状のipEtherResolvRtn関数内ではこの引き数は使われていません．MUXでは，Attach関数(IPの場合ipAttach関数)がポインタif_resolvの初期化の責任を負います．VxWORKSにおけるBSDスタイルのドライバでは通常，ether_output関数がarpresolve関数を直接呼ぶので，if_resolveにNULLがセットされます．

さて，話をipOutput関数に戻しましょう．ipOutput関数は，ENDとNPTの両ドライバ・モデルのパケットを処理します．END/NPTともに，mbufの先頭にデータ・リンク層ヘッダ用の空きを確保(Prepend)するのは同じですが，ENDがmuxAddressForm関数にてデータ・リンク層のヘッダを作成して，フレームがワイヤに出て行ける状態まででき上がるのに対して，NPTでは送り先のデータ・リンク層アドレス(EthernetではMACアドレス)がすでにわかっていたら，それをmbuf先頭の空き領域にコピーします．

送信パケットは，IF_ENQUEUEでインターフェースの送信キューにキューされて，関数ポインタif_start経由でドライバの送信ルーチンで処理されます．BSDの場合はドライバのTxStartup関数が呼ばれるのに対して，MUXでは関数ipTxStartupが呼ばれます．

関数ipTxStartupでは送信キューのパケットをすべて送信するか，送信フレーム・ディスクリプタの空きがなくなるまで送信を試みます．Ethernetデバイス・ドライバのソース・コードはTornadoに同梱されているので，ドライバの送信ルーチンの詳細については，読者にソース・トレースしてもらうことにし，ここでは割愛します．デバイス・ドライバ・レベルの送信処理の内容はEthernetコントローラに依存します．

BSDドライバのTxStartup関数では，mbufから送信フレーム・ディスクリプタへデータ形式を変えてコピーしたりPHYに依存した処理を行うなど，データ処理がmbufベースとPHYベースで混在し，Restart処理もやや複雑です．これに対してMUXのipTxStartup関数ではすべての処理をmbufの単位で行っており，見通しの良い作りになっています．

ENDなら muxSend 関数，NPTなら muxTkSend 関数でパケット送信を試み，PHYがビジーで送信できない場合は muxSend/muxTkSend 関数の返り値が END_ERR_BLOCK となります．この場合は，いったんデキューしたパケットを再度インターフェースの送信キューにマクロ IF_PREPEND でキューし直して ipTxStartup 関数を終了しますが，キューがいっぱいなら，そのパケットを廃棄します（mbuf チェインを free する）．

ENDは ipTxStartup 関数が呼ばれた時点で完全なデータ・リンク層フレームなので処理は至ってシンプルなのですが，NPTに関しては多少複雑な処理が必要となります．関数 ipOutput 内で mbuf 先頭にデータ・リンク層アドレスを追加してから ipTxStartup を呼んでいますが，ipTxStartup ではこのアドレスを一時データにコピーした後，m_adj 関数でこのアドレス追加をキャンセルして（ポインタを元に戻して）います．これはNPTのアーキテクチャを背景にしています．ネットワーク層の ipTxStartup 関数ではデータ・リンク層へのヘッダを操作しませんが，返り値 END_ERR_BLOCK により ipTxStartup 関数が再度呼び出された際に mbuf 先頭がデータ・リンク層のヘッダを指している場合，それを除去（mbuf ポインタの操作）したうえで muxTkSend 関数を呼ぶ必要があるのです．

このデータ・リンク層のアドレスは，muxTkSend 関数への引き数として必要です．

最後に，muxSend/muxTkSend 関数について説明します．muxTkSend 関数がデータ・リンク層のアドレスを作成する点を除いて，二つの Send 関数は基本的に同じです．muxSend 関数はシンプルな作りで，muxBind 関数によってそのインターフェースにOUTPUTフィルタが登録されていたら，これから送信するパケットに対してそのフィルタを実行し，muxSend 関数を終了します．フィルタが登録されていなければ，END_OBJ 内のドライバ関数テーブル NET_FUNC に登録されているドライバの send ルーチンへパケットを渡します．

この send ルーチンで実際何をやるかはPHYに依存しますが，sosend 関数によって当パケットに割り当てられた mbuf はドライバの send ルーチンで解放されます．これでわかるように，VxWORKS TCP/IPプロトコル・スタックでは Socket レイヤからドライバに至るまで，データ・コピーは発生していません．

おわりに

VxWORKS TCP/IPプロトコル・スタックは，BSDをベースに絶え間ない改良と斬新なアイデアを取り込みつつ，組み込み機器向けのTCP/IPプロトコル・スタックとして進化してきました．

今回は，誌面の関係でその一部を解説するだけにとどまりましたが，VxWORKSとそのTCP/IPプロトコル・スタックの魅力が少しでも伝われば幸いです．

VxWORKSネットワークに関する技術的な質問があれば，VxWORKS User's Group（http://www-csg.lbl.gov/vxworks/）に投稿してもらえれば，筆者の知る範囲でお答えしたいと思います．また，友人の一人でもあるJohn Gordon氏のHP blueDonkey.org VxWorks CookBook（http://www.bluedonkey.org/cgibin/twiki/bin/view/Books/WebHome）も参考になると思います．blueDonkey.orgのVxWorks CookBookは読者参加型のHowToサイトです．貢献したいという読者の皆さんはぜひ参加してください．

参考文献
（1）*Tornado User's Guide*, WindRiver
（2）*VxWORKS Programmer's Guide*, WindRiver
（3）*VxWORKS Network Programmer's Guide*, WindRiver
（4）*TCP/IP Illustrated Vol.1*, Stevens, Addison Welsley
（5）*TCP/IP Illustrated Vol.2*, Stevens, Addison Welsley
（6）*UNIX Networking Programming*, Stevens Prentis-Hall
（7）*The Design and Implementation of the 4.4BSD Operating Systems*, McKuisk,Bostic,Karels,Quarterman,AddisonWelsley
（8）*4.4BSD UNIX Networking Internals*, Mike Karels, BSDI,Inc

はまぐち・じゅんいちろう　カリフォルニア州バークレー在住プログラマ
E-mail： inquire2hama@yahoo.co.jp

組み込みプログラミング・ノウハウ入門(第13回)

アクティブ・オブジェクト・モデリングのこころ

順序集合分割法【その1】

藤倉 俊幸

1 はじめに

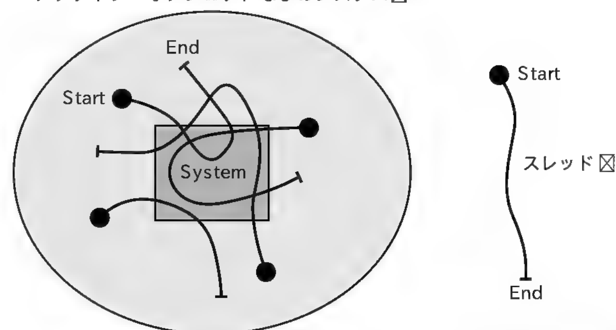
アクティブ・オブジェクトの考えかたは以前からあったが，昔は並列化オブジェクト指向言語と呼ばれる言語仕様の枠の中で議論されていた(1)．今のようにモデリングが重要視される前の話である．

今は，プログラミング言語よりもアプリケーションに近い「モデリング・レベル」でソフトウェアを考える時代になり，特殊な言語を考えなくても，CやC++，Javaでアクティブ・オブジェクトを使えるようになった．プログラミング言語に手を加えるのではなく，動くためのメカニズムとしてのフレームワークを既存の言語で記述して，その上にアプリケーション・モデルを構築するスタイルである．このスタイルでのアクティブ・オブジェクトは，プラットホームからアプリケーション・モデルへスレッド制御が上がってくるための窓口のようなものである．そして，スレッドは基本的にはアクティブ・オブジェクトに閉じ込められてアプリケーション・モデル内を渡り歩くことはない(図1)．このようなアクティブ・オブジェクトを使うことで，MDA(Model Driven Architecture)やxUML(Executable UML)などの動くモデルを作ることが容易になる．

動くモデルを記述するためにUML2.0では，シーケンス図やアクティビティ図が，制御構造を記述できるように拡張される(2)．これらの拡張は手続き的側面が強く，使いかたをまちがうとまったく使いまわしの利かないモデルを作ることを可能にしてしまう．手続き的なモデルは，拡張性が乏しく反復開発の過程で容易にスパゲティ化することが予想される．スパゲティ化したUMLモデルは，ビジュアルなぶん，ただのソース・コードよりも始末が悪い．ソース・コードであれば上から下に読めば良いという流れが存在するが，ビジュアル・モデルはどこから見れば良いのか取り付く島がない．まだUML2.0の制御構造記述に完全対応したツールは存在していないが，既存のMDAツールでさえ，すでにこのような問題に直面することがある．"動く"ことは，良い面もあるが，落とし穴でもあるのだ．

筆者の経験則でいうと，スパゲティ状態に陥る一つの原因として，アクティブ・オブジェクトとパッシブ・オブジェクトの使い分けがうまくできていないことがあげられる．アプリケーション・モデル・レベルでパッシブ・オブジェクトがタスクやスレッドの制御を直接行うとスレッドが渡り歩くモデルになってしまうため，排他制御の問題やプラットホームとの密結合の問題を引き起こす．その一方で，パッシブ・オブジェクトで十分なところにアクティブ・オブジェクトを導入して失敗することもある．たとえばモデルとしてきれいであっても，アクティブ・オブジェクトを使いすぎてリソースを浪費したため，ターゲット環境で動かすことが非現実的になってしまう場合である．

しかし，これらのことは表面的なことであり，もっと本質的な問題は「アプリケーションの並列性などの動的構造を分析段階から抽出し，それを設計に引き渡す方法論がほとんど存在していない」ことである．アクティブ・オブジェクトにはアクティブ・オブジェクト用の方法論が必要である．また，同期メカニズムなど設計の段階で，検証すべきことを実施しないで実装に入ってしまうことも問題である．たとえば，safety(いつまで



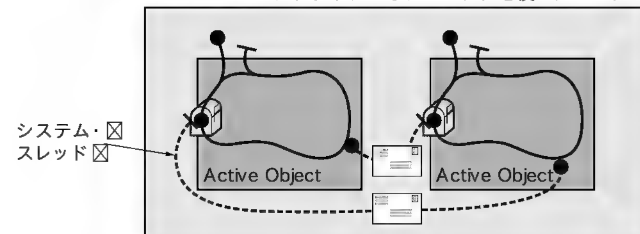


図1 アクティブ・オブジェクトはスレッドの閉じ込め

経っても悪いことは起こらない)とliveness(いつかそのうち良いことが起こる)のような性質は設計段階で解決すべき問題であり，デバッグ段階でソース・コードを直しながらトライ＆エラーで解決できる問題ではない．従来のオブジェクト指向の方法論では，この辺のことはほとんど考慮されていないので，出たとこ勝負のCMMレベル1の世界になっているのが現状である．

2 動的構造の分析

ものには順序がある．実行する順番を変えると別の結果になる場合と，順番を変えても同じ結果になる場合がある．順番を変えても結果が同一の処理は，並行に実行することができる．

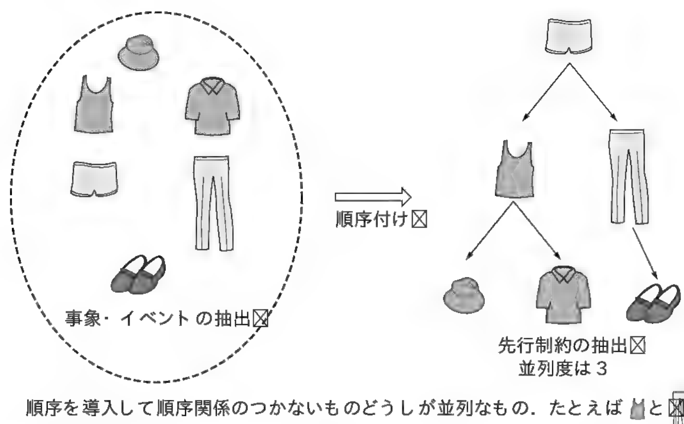


図2　洋服を着る場合の先行制約

並行に実行することのできる処理の数が，設計工程においてタスク数やアクティブ・オブジェクト数の目安になる．したがって，開発対象ドメインに含まれる順番の制約関係を抽出することが必要である．分析段階で，対象ドメインにどのようなオブジェクトがあるか，どのような事象があるか，どのようなイベントがあるかを抽出するだけでなく，これらの間の前後関係も抽出することが重要である．分析段階における動的構造とは，ここではドメインに本質的な並列度や順序制約に関する情報のことをいっている．

● 服を着る場合の分析

たとえば，洋服を着る順番を図2に示す．一般的なオブジェクト指向分析では，図2の左側の事象・イベントの抽出の後は，これらの抽出したものの論理的な関係や実現される機能の獲得に進んでしまい，図2の右側の事象間の前後関係にはあまり注目しない．動的構造は，前後関係を使って右側の図のようにまとめると見えてくる．左側はただの集合だが，右側は順序集合になっている．順序集合は要素の順序関係を使ってハッセ(Hasse)図という図で表現できる．これが図2の右側の図[注1]である．図の意味は次のようになる．

「まずパンツをはく．次はズボンをはくか，ランニングを着る．ズボンとランニングのどちらを先にするかは自由である」…

注1：数学の本ではハッセ図は下から上に描くのが一般的である．ここでは，後でシーケンス図との対応を取るので上下が逆になっている．すなわち，上から下に事象が実行される．



アクティブ・オブジェクト

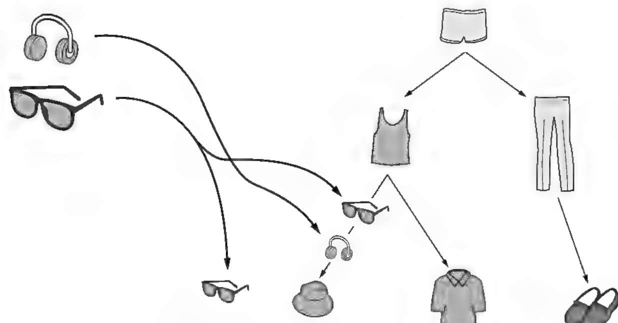
アクティブ・オブジェクトについてはいろいろな定義や解釈があると思うが，ここでは「それ独自のスレッド制御を持っているオブジェクト」の意味で使う．UMLの定義もこのような感じである．JavaのThreadクラスのサブクラスやRunnableインターフェースを実装したクラス，MDAを実現している各種CASEツールの動的なクラスなどがアクティブ・オブジェクトと呼ばれる．RTOSのタスクやスレッド自身もカプセル化などのオブジェクトとしての性質を満たした使いかたをすれば，上記の定義に包含されるのでアクティブ・オブジェクトである．

UMLではメタモデル・クラスのClassがClass::isActiveという属性を持っていて，この属性がtrueならアクティブ・クラスでfalseがパッシブ・クラスになる，という2通りしかない．したがって，スレッドと一対一で対応する形でアクティブ・オブジェクトということばを使ってしまうと，一つのスレッドに複数のアクティブ・オブジェクトを載せられるフレームワークを実現しているMDAツールで使うことばがなくなってしまう．この場合のMDAツールのアクティブ・オブジェクトをリアクティブ・オブジェクトと呼ぶこともある．

ただし，MDAツールにもいろいろあって，リアクティブ・オブジェクトをサポートしていない自称MDAツールもあるので話がややこしくなる．やはり，スレッドはスレッドと呼んでおいたほうが良い．

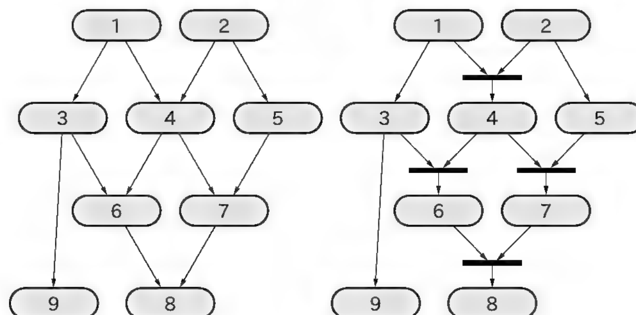
UMLでは，アクティブ・クラスのステレオ・タイプとして<<process>>と<<thread>>があるので，上の意味でのリアクティブ・オブジェクトをサポートしていないMDAツールは，<<process>>か<<thread>>を使用するべきだと思う．「我々は慎重にことばを選んで定義している」とUML作成に参加している某メンバがいっていたが，どんな意味にも取れるように慎重にことばを選んでいるという意見もある．

それでは，ここで扱うアクティブ・オブジェクトとは何かといえば，アクティブ・オブジェクトとリアクティブ・オブジェクト両方を対象にしている．つまり，<<process>>と<<thread>>でないアクティブ・オブジェクトを対象としている．システム全体の視点から見ると，スレッドとの関係が気になるので定義を厳密にしてほしくなるが，アプリケーション・モデルを作る立場では，スレッドもリアクティブ・オブジェクトもとくに区別する必要はない．アクティブ・オブジェクトの制御は下のフレームワークから湧き出してくるようなイメージなので，アプリケーション・モデル内ではほかのオブジェクトから呼び出されなくとも自律的に動くオブジェクトがアクティブ・オブジェクトだと理解すれば問題はない．



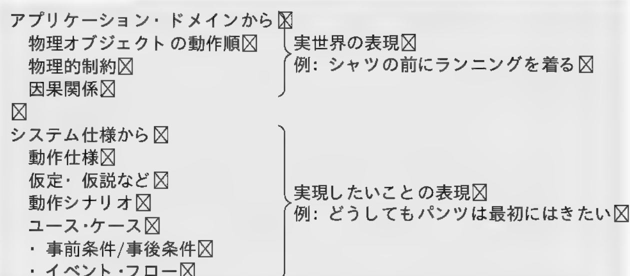
■ヘッドフォンを追加するなら帽子の前
■サングラスは，ヘッドフォンの有無で追加する場所が変わる．
　ヘッドフォンがない場合，サングラスを追加することでシステム全体の並列度が４になる

図３　ヘッドフォンとサングラスを追加する場合



同期情報付き

図４　アクティビティ図の例



アプリケーション・ドメインから
　物理オブジェクトの動作順
　物理的制約
　因果関係
実世界の表現
例：シャツの前にランニングを着る

システム仕様から
　動作仕様
　仮定・仮説など
　動作シナリオ
　ユース・ケース
　・事前条件/事後条件
　・イベント・フロー
実現したいことの表現
例：どうしてもパンツは最初にはきたい

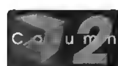
図５　分析される順序の出所

対象ドメインに含まれる順序関係を抽出すると，ズボンとランニングのように順序関係のつかないものが出てくる．この順序関係のつかない事象や処理が並列に実行できるものになる．並列関係は順序関係の否定のような形で出てくると考えるのである．この並列関係は，同時に実行しなければならないというような強い意味ではなく，どんな順番で実行しても良いという意味である．

順序関係については，「パンツが先でズボンが後」，「ズボンが先で靴が後」，であれば「パンツが先で靴が後」というぐあいに推移関係がある．つまり，A→BかつB→CであればA→Cが成立する．しかし，並列関係では推移関係が一般には成り立たない．たとえば，自転車に乗りながら歌を歌える，歌を歌いながら自動車を運転できる，でも，自転車に乗りながら自動車の運転はできないのである．つまり，順序関係にはハッセ図で表現できるような数学的な構造がある．そのため，並列関係自身よりも数学的扱いが容易なので，順序関係を使って対象ドメインに含まれる動的構造を抽出する．

一度順序構造を抽出したら仕様の追加や変更などの際には，図２のようにハッセ図をメンテナンスしていく．たとえば，ヘッドフォンを仕様追加した場合，ハッセ図のどこに入れるべきかを考える．ハッセ図への挿入位置がわかれば，システム全体の並列性への影響を評価することができる．このようにハッセ図は，インクリメンタルに再利用できるので非常に有用である．UMLでは，アクティビティ図を使って表現できる．たとえば，図３のアクティビティ図は図４のようになる．

順序構造は，一般的に図５に示したところから抽出される．大きく分けると，「実世界の物理的制約」と「何をしたいか」というシステム仕様の二つから順序を抽出する．この二つが混乱すると何をやっているのかがわからなくなる．事象の順序はなるべく細かく抽出して，それらが並列で動くことを仮定してシステム環境を分析する．実際は，システムに関わる事象全体が一つのハッセ図になることはまれで，複数のハッセ図になったり，あるいは条件によって起こらない事象などがあるので，部分的



Column 2　プロセス代数の種類

一般にプロセス代数と呼ばれるものは，

- CSP（Communicating Sequential Processes，1978 Hoare）
- CCS（Calculus of Communicating Systems，1980 Milner）
- ACP（Algebra of Communicating Processes，1984 Bergstra）
- ATP（Algebraic Theory of Processes，1988 Hennessy）

の４種類である．表記法はそれぞれ異なる．意味付けは，どれも始めのほうは同じようだが，内部プロセスなどが出てくるあたりから扱いかたが違ってくる．雰囲気的にはACPとATPは代数らしい体系である．CSPとCCSは通信系とか分散系を対象として，1980年代から実用化されるようになり，Occamやトランスピュータ，Lotos，SDLなどの成果があるが，特定の分野以外では最近は忘れられた状態に近いかもしれない．

しかし，1990年代後半から，代入文などの言語的部分を取り除いた第２次CSP，あるいはCCSをベースとしたACSR（Algebra of Communicating Shared Resources）に関する書籍が出版されるようになった．直感的には，互いに通信しながら並行動作をするものから構成されるシステムを記述する際にはプロセス代数が向いている．この，互いに通信しながら並行動作を行うものがアクティブ・オブジェクトと相性が良いので，MDAへの応用が期待される．

ここではプロセス代数の入り口あたりの成果を使って，ステート・マシンの生成を行う．

にステート・マシンで表現することになる．複数のハッセ図になる例として踏切制御を次に取り上げる．

● **踏切制御の分析**

電車の踏切の例を図6に示す．踏切の仕様としては，「電車が接近したら，遮断機を閉じて，電車が通過したら，遮断機を上げる」と記述とできるので，これをシーケンス図で表現して，さらに容易に制御機の動作を記述するステート・マシンまで作成することができる(図7)．MDA環境を使用していれば実行形式まで作ることができる．

しかし，実世界の事象は並列して起こること，外部オブジェクトは独立して勝手に動くことなどを考慮すると，これでは分析が不十分である．どこが不十分かというと，図7のシーケンス図に含まれる順序制約を，順序対に分解して，各順序対の帰属を行うことで明らかになる．実際に順序対を抽出すると図8に示すA～Eの5種類が抽出される．そして，各順序対の帰属を行うと図9のようになる．ここで，P1とP2は物理現象であり制御の対象外である．Sp1とSp2が基本的なセンサとアクチュエータの動作になる．Sp1注2とSp2はソフトウェアで制御できるが，P1とP2は制御できない．そして重要なことは，要求仕様にも外界の物理制約にも帰属できない順序対Dの存在が明らかになる．この踏切の例は単純なのでDの意味はすぐに理解できるが，実際の組み込みシステムの場合にはそうとは限らない．一般には，帰属できない順序対を含めた場合のふるまいと，含めない場合のふるまいを比較することで順序対の意味を明らかにする．どのようにするかというと，帰属がはっきりした各事象の順番対のみからハッセ図を生成して，そのハッセ図からコラム4で述べた方法でステート・マシンを生成する．次に意味のよくわからない順序対を加えたハッセ図から同様にステート・マシンを生成して，これら二つのステート・マシンのふるまいを比較する．

この例の場合，帰属が明らかな順序対のみからハッセ図と状態図を作ると図10のようになる．生成されたステート・マシンは，

　　動作：接近→通過→閉じる→開く

という好ましくない動作を含んでいることがわかる．つまり，「電車が接近して，電車が通過したら，遮断機を閉じて，遮断機を上げる(S1-S2-S3-S4)」という超高速電車と反応の遅い遮断機の場合である．

次に，帰属の明らかでなかった順序対Dをハッセ図に加えてステート・マシンを生成すると図11のようになる．ここでは順序対DをSp3として，システム運用規則，あるいは設置条件のような形でまとめている．図7の場合では，これらの前提条件があいまいなまま制御機のふるまいを決めている．暗黙のうちにつごうの良い仮定をしているといっても良い．この仮定に根拠があれば問題はないが，普通はソフトウェアでは制御できない範囲の動作を，いつの間にか動作の一部として入れてしまうことが多い．この場合，開発対象の動的構造の分析が不十分であり，その結果，システムのテスト段階で不具合が発見されることになる．しかも，分析段階で逃がしたエラーは，ほとんどの場合，非常に発見しにくい不具合になることが多く，発見するまでに多くの工数を必要とする．しかも，市場に出てからの

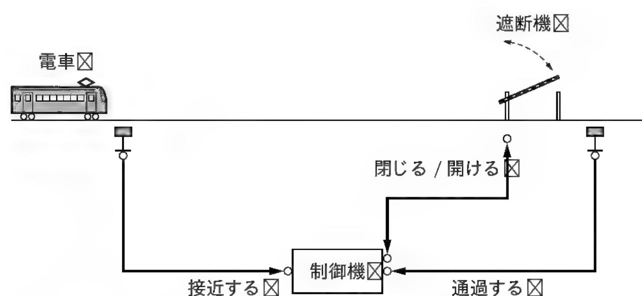


図6　踏切の例

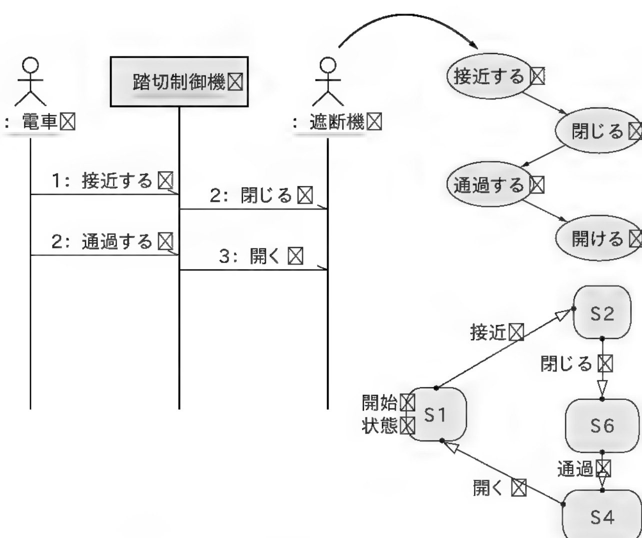


図7　一般的な手法

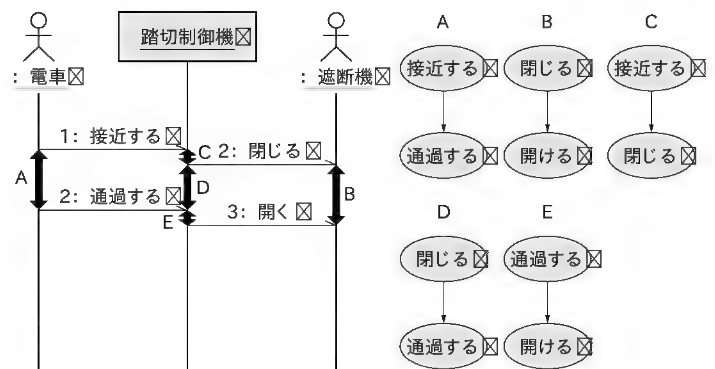


図8　順序対の抽出

注2：図5の説明で「自動車を止める」と補足説明をしたが，正確に自動車について言及するためにはドメイン・オブジェクトとしての自動車に関する事象も加えて議論する必要がある．ここでは簡単にするため自動車オブジェクトは省いてある．

Column 3 ハッセ図

集合の要素間に順序を入れた集合を順序集合と呼ぶ．その順序集合を見やすく表現するために使われるのが，ハッセ図である．

順序集合を視覚的に表現するには矢印を使って要素間の順序を表す．つまり要素が点，関係が矢印の有向グラフである．しかしそうすると，すぐに矢印だらけの図になるので，自分に対する矢印（反射的関係：ここで使用する先行制約では使わない）と推移的に導くことができる矢印を省く．また，矢印の向きを上向きか下向きに決めてしまうことで，矢印の矢を取り除いてただの線分にしてしまうとすっきりとした図を描くことができる．それが，ハッセ図である．ハッセ図や順序集合の性質の詳細については，数学の集合論の適当な入門書を参考にすると良い．二項関係とか半順序，全順序，鎖，反鎖の概念が重要である．数学の本では抽象的すぎて，つらいと思う人には以下の本をすすめる．

- 守屋悦朗，「コンピュータサイエンスのための離散数学」，サイエンス社，ISBN 4-7819-0643-5.
- C. L. Liu，「コンピュータサイエンスのための離散数学入門」，オーム社，ISBN 4-274-13007-X.

クレームや事故で発見されることになる．

図6の例の場合，結局は図7と同様の，

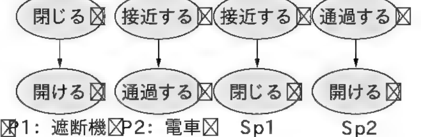
Sp4：電車の接近→遮断機閉じる→電車通過
→遮断機を開ける

を仕様としてシステム全体のステート・マシンを生成したことになった．Sp4では抽出した事象すべてについて順序関係を指定してしまったので，順序から分析できることは何もなくなってしまう．全順序関係であれば並列性を考慮する必要はない．Sp4はシステムが満足すべき性質（safety 性）を記述したものであり，システムの挙動を記述したものではない．実際は，電車オブジェクトを複数走らせるなどして，抽出したハッセ図とステート・マシンからシステム全体の動作を合成して Sp4を満足するかどうかを検証することで，設計フェーズに進むことができる．

電車が複数走る場合の扱いについては別途説明することにして，ここではハッセ図とステート・マシンを利用した動的構造分析の重要性を指摘するだけにして，次節ではハッセ図からどのようにして並列性を抽出し，タスク分割やアクティブ・オブジェクト抽出につなげるかについての説明に移ることにする．

要求仕様書やシステム構想書から Sp4を読み出すだけなら国語の演習問題にすぎない．しっかりとした方法論に基づいた動的構造分析の裏付けが必要である．“しっかりした”という表現はあいまいであるが，少なくとも検証可能でなければならない．アプリケーション・ドメインから抽出すべき動的構造には，ハッセ図で扱える順序のほかに処理時間を考慮したタイミング構造を考えなければならない．処理時間を考慮した結果を図11のSp3のような順序制約として反映できない場合には，この連載の第7回の時間オートマタ(3)や第9回・第10回の時相論理な

アプリケーション・ドメインから
P1：遮断機を閉じる
→開ける
・閉じなければ開けられない
P2：電車が接近する
→通過する
・接近しなければ通過できない



システム仕様から
Sp1：電車が接近→遮断機を閉じる
・safety：悪いことは起こらない
- 自動車を止める
Sp2：電車が通過→遮断機を開ける
・liveness：良いことが起こる
- いつか必ず自動車も踏み切りを通過できる
- 開かずの踏み切りにはならない

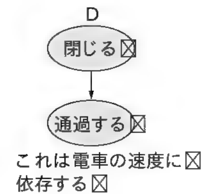


図9　順序対の帰属

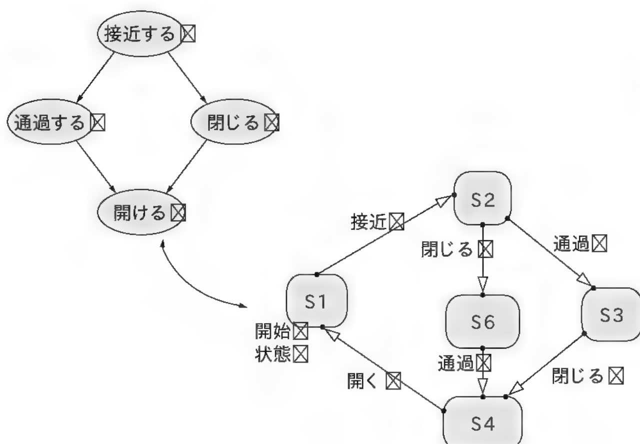


図10　踏切制御の分析-1

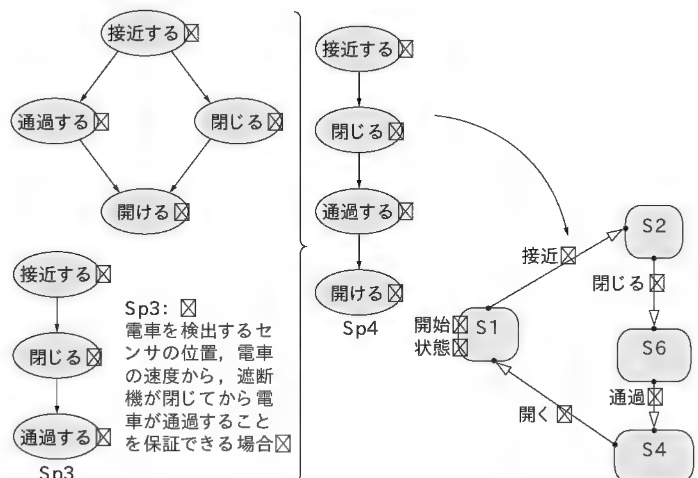


図11　踏切制御の分析-2

どのアプローチが提案されているが，残念ながら研究レベルである．研究レベルではあるが，これらの記事を書いてからもいろいろな成果が報告されている．それらを組み合わせると開発プロセス全体をカバーできそうな状況になりつつある．今回は，まず順序だけを考える．

3 動的構造の設計

動的構造分析の結果得られた，ハッセ図やステート・マシンからどのようにアクティブ・オブジェクトを抽出してふるまいを定義するかについて説明する．小さなハッセ図やステート・マシンが複数得られた場合は，従来のオブジェクト指向分析（静的な分析）の結果から得られた分析オブジェクトとの整合性を取りながら並列化オペレータで合成することで，任意の粒度のふるまいを定義することができる．問題になるのは並列性を含んだハッセ図が得られてしまった場合の分割方法である．普通は，ハッセ図の分割が必要になる．

具体例として図12に示すようなハッセ図が得られたとする．この図の意味は，9種類の処理があり，それぞれの処理は割り込みなどの外部イベントにより起動される．ただし，実際に処理を実施するためにはハッセ図で示された順序制約がある．たとえば処理4は，処理4を起動する外部イベントを受信しても処理1と処理2が終了していないと実施できない．一般的な設計手順は図13のようになる．

● 分割しない場合

まず最初に考えるべきことは，分割するか，しないかである．アクティブ・オブジェクト・モデリングでは，並行実行の単位がアクティブ・オブジェクトになる．別々のアクティブ・オブジェクトに分割しておけば，それぞれを別々のスレッドに割り当てることでRTOSレベルの並行性を利用した実行が可能になる．一つのアクティブ・オブジェクトの中で利用できる並行性は，UMLで規定された動作セマンティクスであるRun to Completionになる．つまり，そのアクティブ・オブジェクトが持っているステート・マシンの遷移ごとの並行性になる．つまり，プリエンプティブな動作はできなくなる．プリエンプションを利用できないとCPUの使用効率が悪くなり，デッドラインを守れなくなる可能性がある．

したがって，普通は分割する方向で設計を進めるが，RTOS

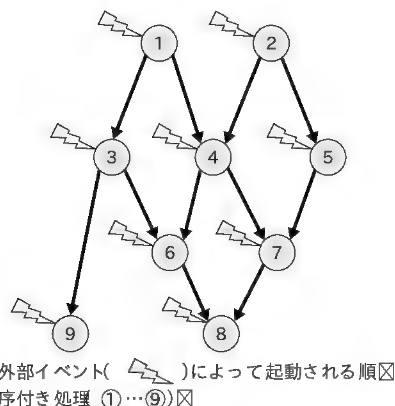


図12 サンプルのハッセ図
外部イベント（ ）によって起動される順序付き処理（①…⑨）

・分析結果から設計方針を決める
　- 並列度が3だからアクティブ・オブジェクト　3個
　- RTOSは使わないのでアクティブ・オブジェクト　1個
・ハッセ図をアクティブ・オブジェクトの数に分割する
・アクティブ・オブジェクトのインターフェースを決める
　- 非同期メッセージの導入
・各アクティブ・オブジェクトの状態マシンを生成する
　- 全実効パスを網羅する逐次プロセスの直積作成
　　・パブリック・ドメイン・ソフトを利用できる
　　　- 例 http://www.doc.ic.ac.uk/~jun/book/ltsa/LTSA.html
　　　　LTSA(Labelled Transition System Analyser)

図13 設計手順



Column 4 ステート・マシンの生成法

図5から図7では踏切のふるまいを状態図で表現した．この際のハッセ図は，そのままCSPのプロセスでも表現できる．たとえばP1：遮断機は，

P1＝閉じる→開ける→P1

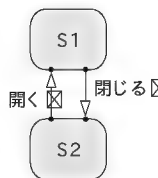
となる．このまま状態図でも表現できる（図A）．

繰り返さない場合は，

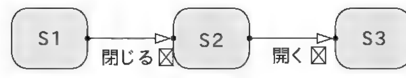
P1＝閉じる→開ける→STOP

となる．このときの状態図は，図Bのようになる．

ハッセ図からプロセス式，さらに状態図と変換することができる．そして，プロセス式を並列化オペレータで結合した複合プロセスは，状態図では個々の状態図の直積によって表現できる．状態図の直積の作りかたについては，この連載の第7回の時間オートマタで説明した．実際の並列化ステート・マシンを生成するには，第7回で紹介したkronosやLTSAなどのオープン・ソフトを使用することができる．つまり，プロセス式さえ書ければそれを組み合わせてステート・マシンはだれでも生成できるのである．複雑なハッセ図の場合には，一つのハッセ図から複数のプロセス式が作られるのでそれを並列化して一つのステート・マシンにする．複数のステート・マシンにする場合には，並列化するプロセスを組み合わせればよい．



図A P1の状態図



図B 繰り返しをしない場合のP1の状態図

を使用しないことがすでに決まっている場合には分割してもプリエンプションは利用できないので，分割をせずに一つのアクティブ・オブジェクトにする場合もある．

リソース制約の厳しい環境では分割しない選択もありうる．分割しない場合は，ハッセ図全体を単独で処理する　ステート・マシンの生成を行う．

ステート・マシンの生成は，図13に示したURLからダウンロード可能なLTSA(Labelled Transition System Analyser)のようなツールを使用すると便利である．

LTSAを使用する場合は，プロセス代数のCSPによるプロセス表現を生成し，それを並列化することでステート・マシンを生成する．分析段階でシステムの挙動を調べるためにステート・マシンを生成した方法とまったく同様である．ハッセ図からそれに対応するステート・マシンを生成するには，ハッセ図の各パスを通るシーケンシャルプロセスを定義する．図14のA0からA5がCSPプロセスの定義である．そして，これらのプロセスを並列化オペレータ(||)で結合して複合プロセスPROGを定義する．これらの定義をLTSAに読み込んでコンパイル・合成を行うとステート・マシンが生成される．生成されたステート・マシンを図15に示す．このステート・マシンの各遷移の中から対応する処理関数を呼び出すようにする．たとえば，a1遷移から`act1()`を呼び出す．かなり複雑なステート・マシンになるが，LTSA上でデッドロックやライブロックなどが起きないことを検証済みのステート・マシンなので，単純なタイプ・ミス以外のバグはないはずである．したがってテストはタイプ・ミスをチェックする程度で良い．このあたりがプロセス代数を使用する形式的手法の良いところである．

分割をしない場合では，開発者が頭を使わなくてはならない作業は，図12のハッセ図を作るところだけである．つまり，人間が行うのは分析だけで，後は機械的な作業なのでだれがやっても同一のソフトウェアができあがる．コンパイラが生成したオブジェクト・ファイルが中間生成物であるように，設計モデルから自動生成したソース・コードも中間生成物となり，さらに形式手法を使えば設計モデルさえも中間生成物と成り果てて，メンテナンスすべきものは分析モデルだけになる日がくるかもしれない．

● 分割する場合

分割する場合については，非常に重要なので，次回説明する．普通は，全体を部分に分割することで複雑度を下げることができる．しかし，動的構造については分割することで複雑度を上げてしまう場合がある．

オブジェクト指向では役割に基づいてオブジェクトを抽出する．動的構造を無視して役割のみでアクティブ・オブジェクトを抽出すると，ほとんどの場合には複雑度を上げてしまう．組み込み分野でオブジェクト指向が普及しないのは，あるいは成功例が少ないのはこのためである．図12程度のハッセ図の場合でさえ，分割に失敗すると図15よりも複雑なふるまいの記

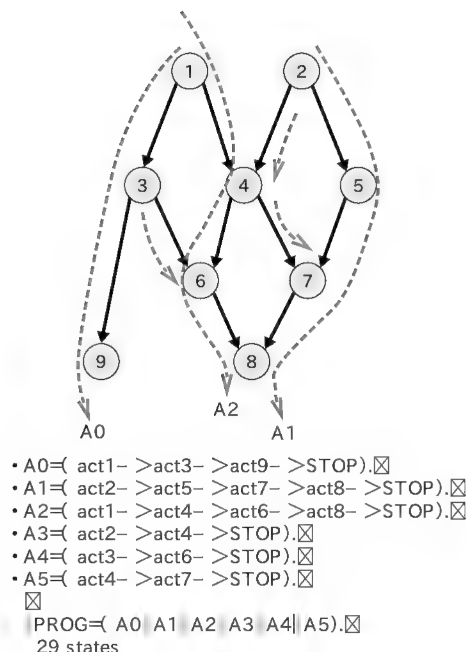


図14　分割しない場合のステート・マシン生成

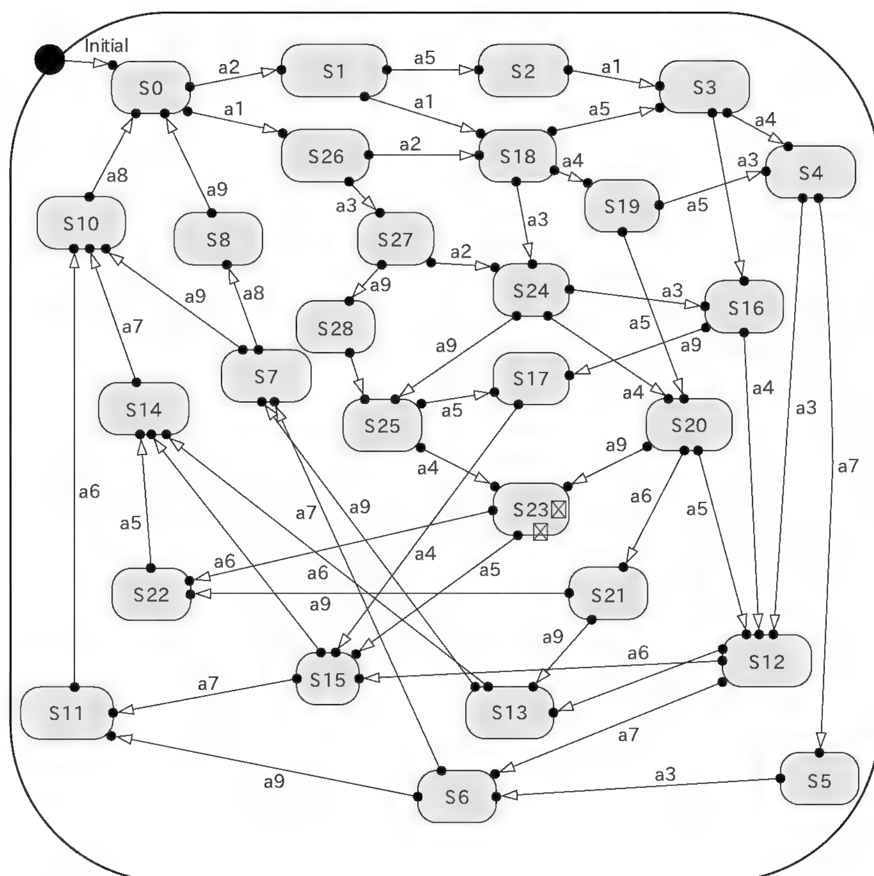


図15　生成されたステート・マシン

述になってしまう．ふるまいが複雑になると，使用するリソースも増えるし，必要なテスト・ケースも増え，開発期間が長くなり，バグも残りやすくなる．つまり，分割しなければ良かったということになる．いい換えるとオブジェクト指向を使わなければ良かったということでもある．しかし，動的構造に基づいて分割すれば，複雑度を下げることができるのである．

4 実装とテスト

生成したアクティブ・オブジェクトをどのように動作させるかについては，使用するMDAツールに依存する．一般的なMDAツールにより生成される実行形式の構造を図16に示す．アプリケーション・モデルの下にアクティブ・オブジェクトをドライブするレイヤがある．この部分をRose Real-TimeではTargetRTSと呼ぶランタイム・フレームワークで構成する[注3]．一般的には，アクティブ・オブジェクト間の通信方法としてメッセージ・キューを使用したり，遅延関数呼び出しを利用したりしてスレッドを分離する．スレッドを分離することでスレッドがモデルの中を渡り歩かないようにしている．その下にRTOSがある．シングル・スレッド用のフレームワークがあれば，RTOSなしで実装することも一般に可能である．

図12のステート・マシンをアクティブ・オブジェクトではなく，パッシブ・オブジェクトのパブリック・オペレーションとしてコード生成する機能があればランタイム・フレームワークも使用しないで，たとえば割り込みルーチンからダイレクトにパブリック・オペレーションを呼び出すアーキテクチャで実装することも可能である．Rose Real-Timeの場合では，パッシブ・クラスでコード生成すると，`a1()`から`a9()`というパブリック・オペレーションを持ったクラスが生成されるので，それぞれの外部割り込みを検出するモジュールから対応するトリガ関数`a1()`から`a9()`を呼び出せばステート・マシンを駆動させることができる．トリガ関数の中からは各イベントに対応した処理関数を呼び出す．

この応用例として，すでに各イベントの処理関数はソフト

注3：ほかのMDAツールも似たり寄ったりである．

Column 5 LTSAツール

LTSAツールは，`http://www.doc.ic.ac.uk/~jnm/book/ltsa/LTSA.html`からダウンロードできる，並行プログラムを検証するためのツールである．使いかたは，Jeff Mageeの本（図C）に詳しく説明してある．実装言語としてJavaを使っているが，MDAツールを使うのであればLTSAツールのモデルをそのまま言語から独立して利用できる．

たとえば，踏切問題の遮断機と電車の動きは，以下のようにモデリングし，

```
P1 = (close->open->P1).
P2 = (detected->pass->P2).
||Sys1 = (P1||P2).
```

複合プロセス`Sys1`をステート・マシン表示にすると，図Dのようなステート・マシンを生成してくれる．これは，遮断機と電車がまったく独立に動いた場合のシステム全体の動きを表している．

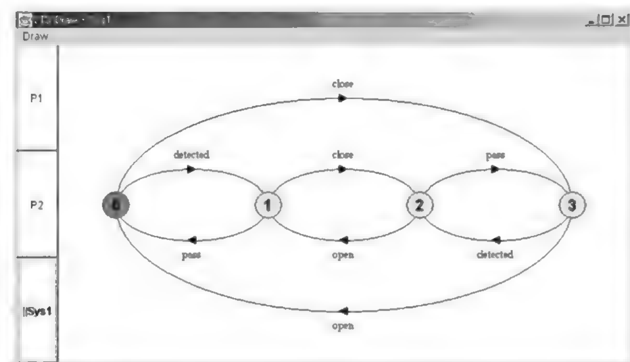
このステート・マシンにシステム仕様の`S1`と`S2`を以下のように追加して，

```
S1 = (detected->close->S1).
S2 = (pass->open->S2).
||Sys2 = (Sys1||S1||S2).
```

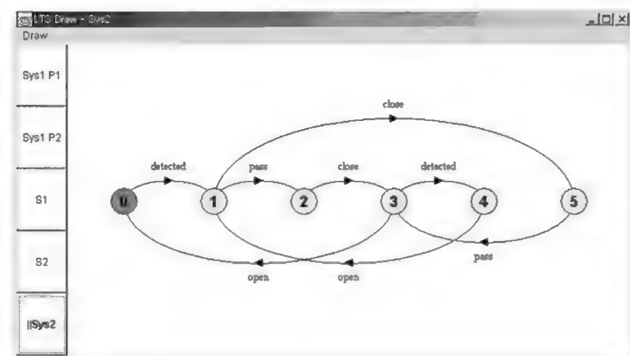
再度，ステート・マシンを生成すると図Eのようになる．このステート・マシンは，本文の図10のステート・マシンに対応する．



図C
Jeff Magee, "Concurrency: State Models & Java Programs"



図D　複合プロセスSys1のステート・マシン表示



図E　再度生成したステート・マシン表示

ウェア資産として存在しているが，今まで動的構造分析を実施していなかったのでsafetyやlivenessを満足しているかどうか心配な場合には，動的構造分析を今から実施して分析結果を反映させるパッシブ・クラスのステート・マシンを生成してイベント入力レイヤとアプリケーション・レイヤの間に挿入すれば，今まで入力イベント順により誤動作していたシステムを再生することができる(図17)．あるいは，挿入した層でログを取ることで動作の解析を行うなどの対策を取ることができる．かなりスパゲティ化したレガシ・ソフトウェアでも入力部分にパッチを当てるくらいはできることが多い．正常系は動くが例外系を付け足していくうちにわけがわからなくなったソフトウェアで，イベント順などの動的部分に不具合があるような場合には，ここで紹介したステート・マシンによるパッチは効果がある．ここで紹介したステート・マシンには動的な例外系を正常系に変換して上位のアプリケーション・レイヤにインターフェースする機能がある．イベント・シーケンスが乱れた場合，たとえばイベント4がイベント1やイベント2よりも先に到着した場合などにどうするかは仕様の問題であるが，イベントを保留する必要がなければ，フレームワークなしのステート・マシン・コード・レベルで対応可能である．イベントを条件が揃うまで保留しなければならない．たとえば，イベント4が早すぎた場合には保留しておいて，イベント1とイベント2がそろってからイ

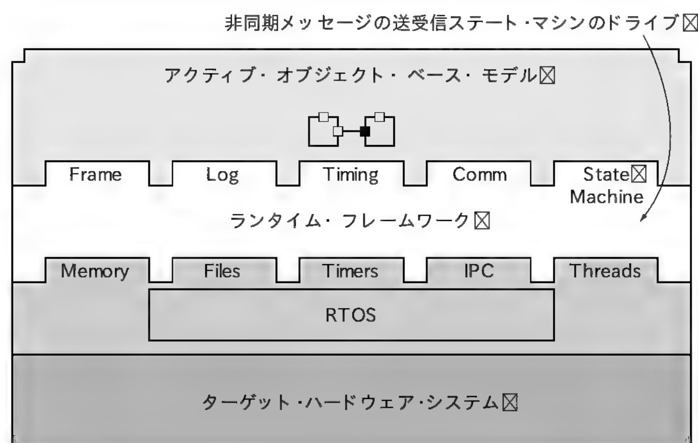


図16　一般的なMDAツールにより生成される実行形式の構造

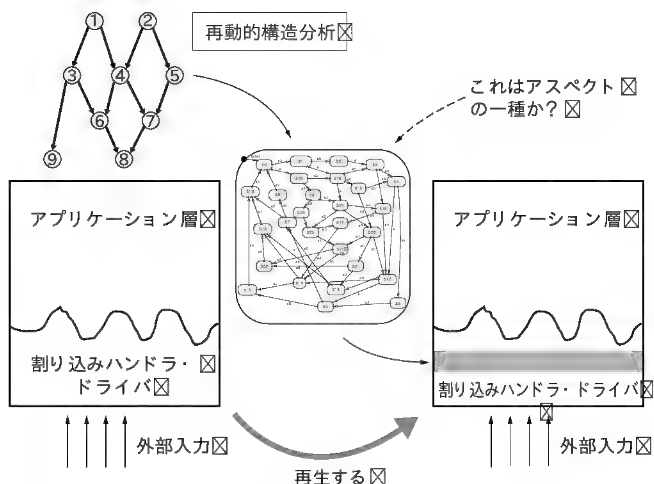


図17　レガシ・ソフトウェアの再生
どのような実装を行うにしても，人間が考える部分は分析工程が中心であり，設計から実装の部分はパターン化されたアプローチが可能で，属人性を排除した開発が可能になる．

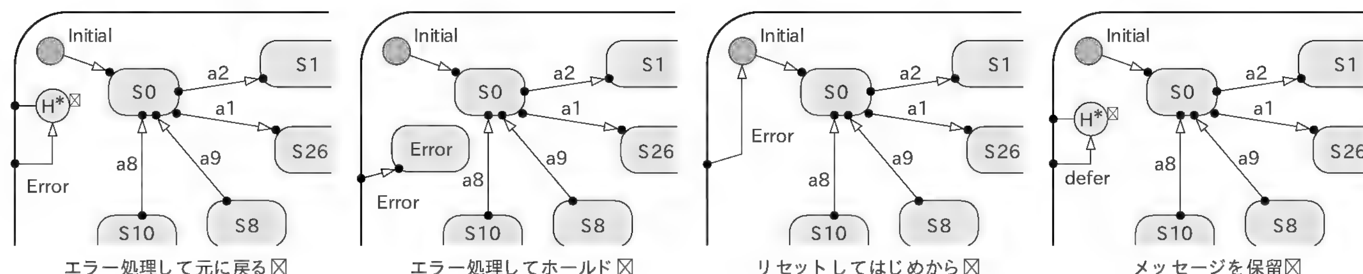


図18　エラー処理

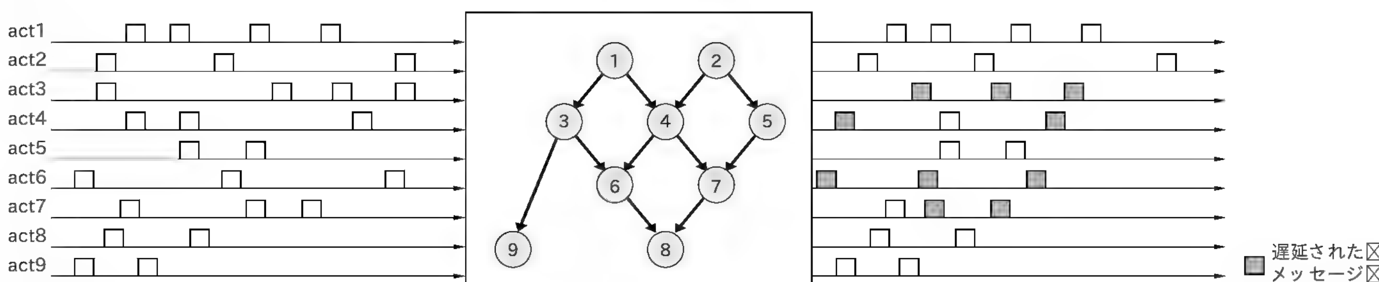


図19　メッセージを保留した場合の動作例



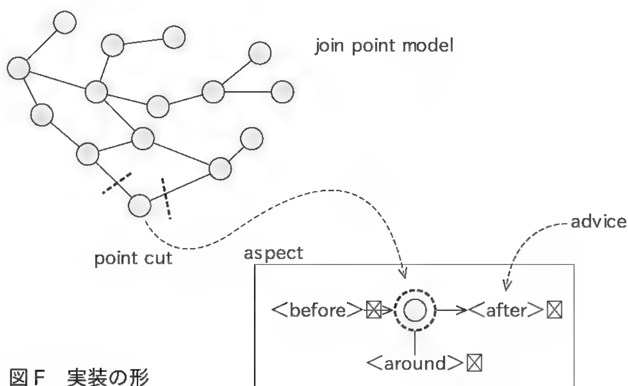
アスペクト指向

オブジェクト指向を補う技術としてアスペクト指向が注目されてきた．サブジェクト指向とかアダプティブ・プログラミング，コンポジション・フィルタなどがアスペクト指向として収斂(しゅうれん)してきた．

オブジェクト指向は，データ構造へのアクセスを中心としてクラスを構成してプログラムを構築するので，仕様を与えられたとき，クラスへの分割を試みてプログラムを作っていく．しかし，実際のプログラムはデータ構造へのアクセスだけが問題なのではなく，ここで扱っているような同期の問題や並列性の問題がある．つまり，いろいろな側面（aspect）をプログラムは持っている．オブジェクト指向では，データ構造の側面を中心としてプログラムの分割が行われる．アスペクト指向的ないいかたをすると，支配的な分割（dominant decomposition）が，データを関心事（concern）としてクラスによって行われる，という感じになる．そうすると，組み込みソフトで重要になる並列性などの関心事は，プログラム全体に分散されてしまう．これを横断的関心事という．このような横断的関心事（crosscutting concern）をアスペクトとしてモジュール化しようというのがアスペクト指向である．一度分断されてしまった関心事を，ソフトウェア・モジュールとしてまとめて，スパゲティ化したプログラムを再生することができる．本文で紹介したステート・マシン・パッチと目的は似ている．また，並列構造を関心事として論理構造から分離しているところも似ている．しかし，実現のアプローチは異なる．

プログラムの表現方法はいろいろある．たとえば，関数呼び出しの連続と見るとか，変数アクセスの連続と見るか，などがある．関数呼び出しに注目すると，関数の呼び出しと入り口とをjoin pointとしてプログラムがつながってできていると考えられる．このような見かたをjoin point modelと呼ぶ．関数のレベルで見ているので支配的だったクラスの枠組みは取り払われている．ここで，関心のあるjoin pointを取り出してくる．このことをpoint cutするという．たとえば，同一のデバイスにアクセスしている関数で排他制御しなければならないものをpoint cutして抽出する．そして，point cutした関数に入る前にセマフォをtakeして，関数から抜けたらセマフォをgiveするようなコードを付け加える．このようなコードをアドバイス（advice）と呼ぶ．point cutとadviceをまとめてaspectとして記述するのがアスペクト指向プログラミングである．このようにすれば，プログラム全体に分断されてしまった排他制御機構を1か所で記述できるので，セマフォIDの付け間違いなどはなくなるし，プログラムも読みやすくなる．また，アスペクトはすでに動作しているプログラムに付け加えることができるので，もつれ合ったプログラムの再生にも使える．

実装の形としては，join pointにフックを仕込んでadviceを実行するようにアスペクト・コンパイラが展開する．そのため，かならずしもソース・ファイルは不要で，Javaを対象としたAspectJではJavaのクラス・ファイルがあれば良い．C/C++の場合は，アスペクト・コンパイラがaspectを仕込んだC++ソース・コードを生成する．ソース・コードに展開するのであれば，上記の排他制御機構の場合，マクロ展開で今まで対応してきた人も多いと思うが，マクロ展開だと自分であちらこちらのソース・コードに自分でマクロを仕込まなければならない．しかし，aspectを利用すればpoint cutを使えるので1か所にまとめて記述できる（図F）．



図F　実装の形

ベント4の処理を行わなければならない．このような保留やキューイングを使用するケースでは，MDAフレームワークやRTOSの機能などを使用することが多い（図18，図19，p.145）．

図15のステート・マシンをハンド・コーディングするのは少しきついが不可能ではないので，MDAツールなしで実装することももちろん可能である．どのような実装を行うにしても，人間が考える部分は分析工程が中心であり，設計から実装の部分はパターン化されたアプローチが可能で，属人性を排除した開発が可能になる．

まとめ

順序集合分割法は，オブジェクト指向とプロセス代数を統合した方法論でありアクティブ・オブジェクトを使用するRoseRTのようなMDAツールと非常に相性が良い．事象の順序集合を抽出する分析工程でほとんどすべてが決まってしまう．今回は，順序集合分割法の導入として，動的構造分析の概要と分割しないで実装する方法を紹介した．次回は分割方法の説明をする．

参考文献

(1) 石川 裕，所 真理雄；オブジェクト指向並行プログラミング言語，情報処理，29，4，pp.325-333，（Apr. 1988）のActor.
(2) Luciano Lavagno, Grant Martin, Bran Selic；“*UML for Real : Design of Embedded Real-Time Systems*”，Kluwer AP, 2003.
(3) 藤倉 俊幸；リアルタイムシステム/マルチタスクシステムの徹底研究，TECHI vol. 15，CQ出版，2003.

ふじくら・としゆき　IBM SWG/ラショナルソフトウェア

小型・軽量でSDメモリ・カードと互換性もある

SDIOカード開発入門

第5回 SDIOカード設計事例(後編)

八十島 広至

はじめに

前回(2月号掲載)では，開発ボードCG100-EDKをベースにしたSDIOカード・プロトタイプの設計開発について解説した．以降のステップとして，SDIOカード・プロトタイプの動作確認，ドライバ・ソフトウェアの開発およびテスト/評価を行い，最終的にカード形状のSDIOカード試作段階に進むことになる．

一般的に，SDIOカード・プロトタイプの動作確認時に必要なドライバ・ソフトウェアは，ハードウェア的な機能やパフォーマンスの限界を確認できるレベルの完成度を最低限確保し，デバイス・ドライバの機能向上やパフォーマンス調整は，最終段階のSDIOカードの試作と並行して行うのが現実的である(図1)．

SDIOカード・プロトタイプのハードウェア動作を確認するときに使用するSDIOカード開発環境ツールSD-IDE(シイガイズ製，図2)について解説する．動作確認時に発見される不具合は，おもにハードウェアに起因するものになる．逆に，この動作確認をパスしたSDIOカード・プロトタイプは，ハードウェア的な基本機能は信頼できるものとなる．

次のステップはデバイス・ドライバの開発およびデバッグであるが，その際のプラットホームとしては動作確認をパスしたSDIOカード・プロトタイプを活用することができる．すなわち，ここでの不具合は，おもにデバイス・ドライバに起因すると考えられるからである．

しかし，実際の開発においては，デバイス・ドライバを使うことによってハードウェアの不具合が初めて明らかになるケースも決して少なくない．しかも最初は，デバイス・ドライバかハードウェアのどちらに原因があるのか特定するのに時間がかかり，両方に不具合が存在する場合もあるのが一般的である．

さらに，ハードウェア設計とデバイス・ドライバ設計を別の人，あるいは別のチームが担当する場合は，おたがいに相手の

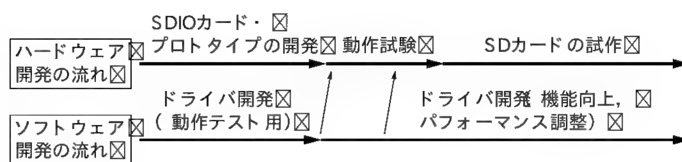


図1 SDIOカード開発の流れ

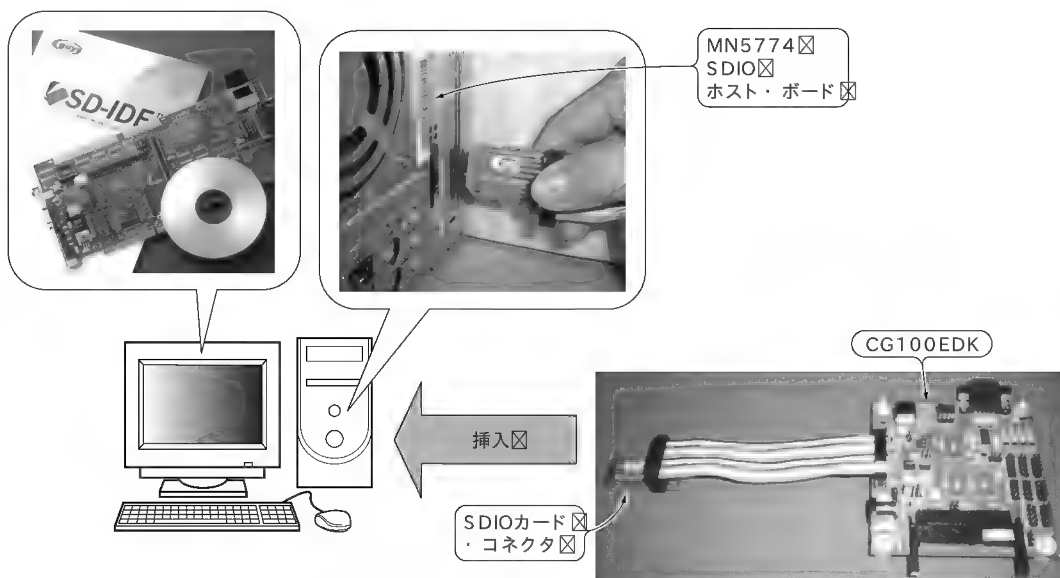


図2 SDIOカード開発環境ツール

注：SD-IDEは，SD-IDEソフトウェアと開発ボードCG100EDKで構成されており，別売はしない

設計の詳細までは理解していない場合が多いことから，開発プロジェクト全体が複雑化して進捗に支障をきたす例が多々発生する．そのような場合でも，SD-IDEに戻ってハードウェアの動作確認にもれがなかったかを再チェックしたり，問題となっている部分を単純化して，不具合部分を再現すれば，ハードウェアに起因する不具合であるのか，ソフトウェアに起因する不具合であるのかを見極めることが簡単になる．

このように，SD-IDEを活用することにより，ハードウェア設計者とソフトウェア設計者がいっしょに作業できる共通のプラットホームを提供することができ，相互の設計の理解とSDIO規格の理解が自然に深まり，デバッグ作業を加速することができると考える．

SD-IDEに関する詳細情報は，次のURLを参照してほしい．

http://www.c-guys.jp/

SDIOデバイス・ドライバの開発

SDIOカード・プロトタイプのハードウェアの動作を確認する前に，SDIOドライバについて触れておく．SDIOドライバは，SDIOホスト・コントローラを制御するSDIOホスト部分（以下，SDIOホスト・ドライバと呼ぶ）とSDIOカードを制御するSDIOカード部分（以下，SDIOクライアント・ドライバと呼ぶ）から構成されるのが普通である（図3）．

SDIOカードを設計するということは，通常はSDIOカードのハードウェアとSDIOクライアント・ドライバを設計するという意味になる．SDIOクライアント・ドライバは，図3のようにOSとSDIOホスト・ドライバに依存するので，これから設計するカードをどのSDIOホスト機器で動作させるのか，また，そのSDIOホスト機器にはどのOSが搭載されているのかなどを事前に考慮して，そのターゲットに合わせた開発が必要になる．

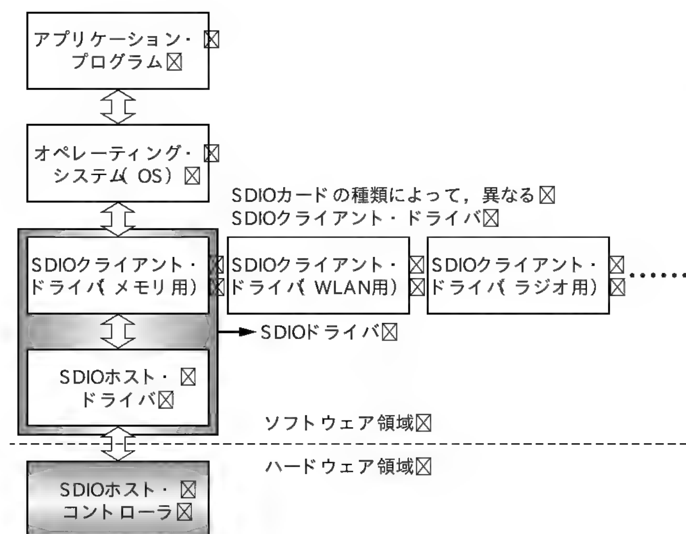


図3　SDIOのドライバ構成

現在，SDIOスロットをもつホスト機器の代表格がPDA（個人用携帯情報端末）である．PDAといっても，さまざまなOSやハードウェア・アーキテクチャがあるが，Windows CE系やPocket PC系では，SDIO Now!（BSquare社，Appendix（p.156～）参照）が事実上，標準的なSDIOホスト・ドライバとして採用されている．Windows CE系やPocket PC系のPDAをターゲットにする場合は，さまざまな機器への互換性を確保するうえで，BSquare社製SDIOホスト・ドライバに対応させた開発が実用的だと考える．詳しくは，次のURLを参照してほしい．

http://www.bsquare.co.jp/products/SDIO/SDIO.htm

Palm系では，機器によって状況は異なるが，SDIOをサポートしている機器の場合は，Palm OSにSDIOホスト・ドライバが用意されている．詳しくは，次のURLを参照してほしい．

http://www.palmos.com/dev/tools/

Linux系では，組み込みLinux系のソフトウェアハウスがSDIOホスト・ドライバを開発しているが，標準としての地位を確立しているものはないようである．また，オープン·ソースになっているものもない．そのほかのOS，μITRON系なども，Linux系と同じような状況である．特にLinux系では，OSと同じライセンス形態（GPL）のSDIOホスト・ドライバの公開が望まれる．Linux系に限る話ではないが，結局，OSにSDIOホスト・ドライバが用意されるか，少なくとも，あるSDIOホスト・ドライバが標準と認知されるまでは，ケース・バイ・ケースで対応する必要があり，場合によってはSDIOクライアント・ドライバと同時にSDIOホスト・ドライバも新規に用意しなければならない．

今後，読者がSDIOホスト・ドライバを開発する際は，ぜひ，標準的なものとして広まるように機能，パフォーマンス，ライセンス形態について考慮していただきたい．SDIOの普及のためには，このドライバ・ソフトウェアの標準化と整備が鍵であり，SDアソシエーションでも早くから活動を行っているが，より多くの読者諸氏の理解と協力が必要である．

SD-IDEによる動作テスト

それでは，SD-IDEによる動作テストを始める．今回は，Windows XPベースのSD-IDEを使って解説する．

まずSD-IDEの箱を開いてみる．中には，解説書（ユーザ・マニュアル）とCD-ROM，SD Host PCIボード，CG100-EDK，SD Plugが入っている．SD Host PCIボードを手持ちのPCの空きスロットに挿し（写真1），PCを立ち上げるとドライバのインストールを促す画面が出てくるので，あとは画面に出てくる指示に従ってインストール作業を進めればよい．

インストールが終了すると，SD-IDEのアイコンがデスクトップ（図4）に表示される．このアイコンは，動作テスト用プログラムを示すものである．SD-IDEはハードウェアも含めた開発

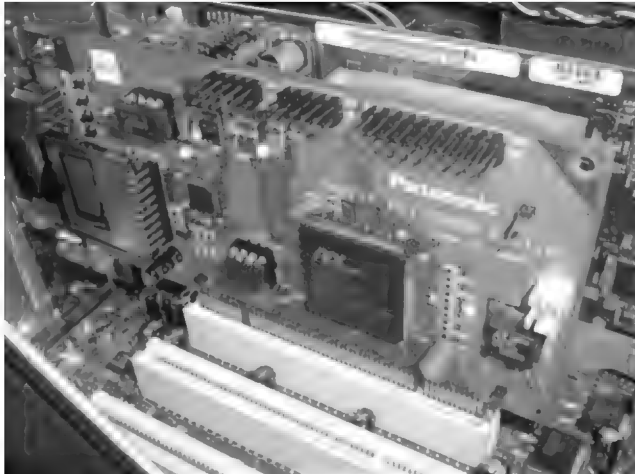


写真1　SD Host PCIボードをPCのスロットに挿す

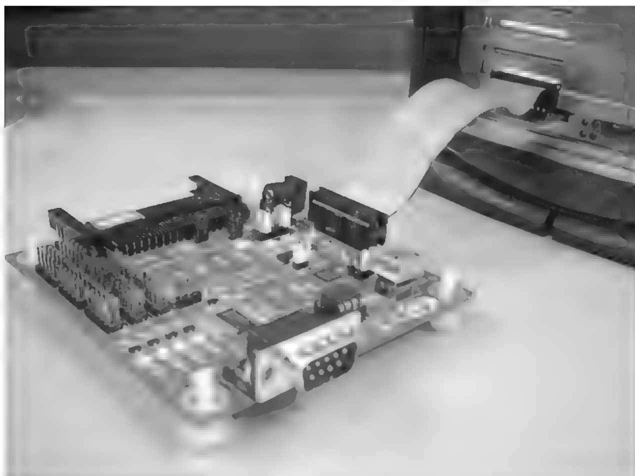


写真2　CG100-EDKをSP-plugを介してPCのSDIOスロットに挿入する



図4　SD Hostのインストール後の画面



図5
SD-IDEのオープニング画面

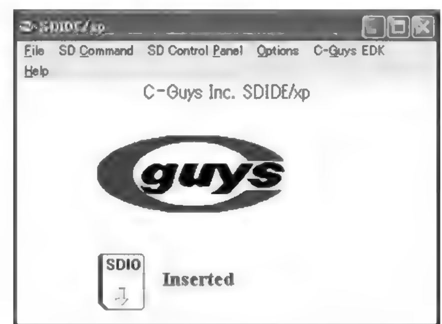


図6
SD-IDEウィンドウにカード挿入のメッセージが出る

環境全体のことを意味しているが，以下，本稿では特に断らないかぎり，動作テスト用プログラムのことをSD-IDEと呼ぶ．

デスクトップ上のSD-IDEアイコンをマウスの左ボタンでダブル・クリックすると，SD-IDEのオープニング画面（図5）が開く．この状態で，CG100-EDKをバス・パワーに設定し，SD-Plugを介してSD-IDEプログラムが載っているPCのSDIOスロットに挿入する（写真2）．すると，SD-IDEのウィンドウの下のほうにカードが挿入されたことを示すメッセージが現れる（図6）．

ところで，SDIOスロットにはマルチメディア・カード，SDメモリ・カード，およびSDIOカードの3種類を挿すことができるが，カードが挿されたあとの初期化シーケンスは，図7のように，最初にマルチメディア・カードかどうかをテストし，次にSDメモリ/SDIOカードであるかどうかをテストする．

その次に，動作電圧範囲を調べ，コンボ・カード（SDメモリ機能を併せもつSDIOカード）の場合は，いったんSDメモリ機能部を初期化して，さらにSDメモリ機能部の動作電圧範囲を調べる．これらのチェックをパスしたら，RCA（Relative Card Address）を割り当て，CMD7でカードを選択状態にする．詳細は，SDIO規格を参照してほしい．SDIO規格の概略版は，次のURLより入手できる．詳細規格の入手には，SDアソシエーションへのメンバ登録が必要になる．

http://www.sdcard.org/

なお，図7のフローチャートは一般的なSDIOホスト・ドライバの初期化シーケンスを示している．ここでは最初のCMD0，CMD1でマルチメディア・カードかどうかをテストしているが，SDIO規格ではここはスキップされて，最初にCMD5が発行される．そして，CMD5にもACMD41にもレスポンスが返らな

電源ON
CMD0を発行
CMD1を発行
レスポンス — あり → マルチメディア・カード用の処理
なし
CMD5(Arg=0)を発行
レスポンス — なし → SDメモリ・カード用の処理
あり
レスポンスからファンクションの数とメモリ・サポートの有無情報を得る
電圧確認 — 電圧NG → SDIOカードを無効にするコマンドを発行
電圧OK
CMD5(Arg=使用電圧)を発行
レスポンス — 準備未完了 / なし → SDIOカードを無効にするコマンドを発行
準備完了
コンボ? — メモリはなし
メモリ・サポート
CMD0を発行
ACMD41(Arg=0)を発行
電圧確認 — 電圧NG → SDIOカードを無効にするコマンドを発行
電圧OK
ACMD41(Arg=使用電圧)を発行
レスポンス — 準備未完了 / なし → SDIOカードを無効にするコマンドを発行
準備完了
CMD2を発行
CMD3を発行〔RCA(相対カード・アドレス)取得〕
CMD7(Arg=RCA)を発行(SDIOカード選択)
通常動作へ

図7　初期化シーケンス

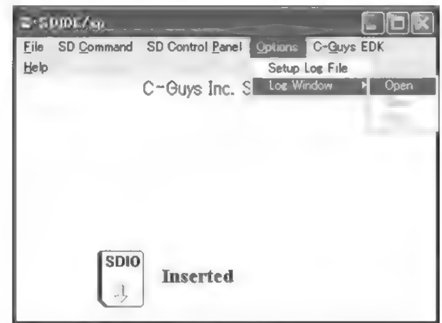


図8
Openをクリック

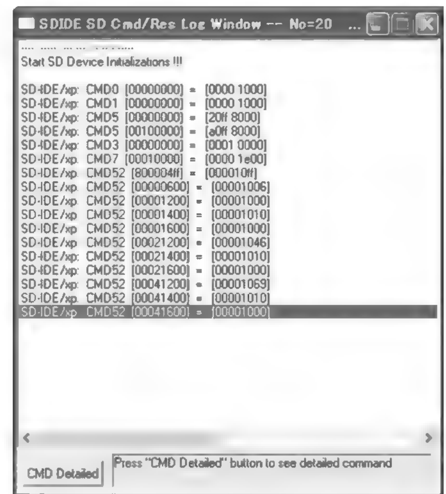


図9
ログのウィンドウを表示

かったときにマルチメディア・カードだと判断するという流れになっている．

SD-IDEでは初期化シーケンスは自動的に実行されるので，ユーザがいちいち初期化シーケンスに従ってコマンドを発行する必要はなく，すぐにユーザが設計したハードウェアへのアクセスを開始できる．すなわち，図6の時点で初期化シーケンスは終了している．なお，初期化シーケンスで発行されるコマンドは，SDIOカード・コントローラ・チップCG100で処理されるので，この段階では不具合は起きないはずである．

CG100を使っていない場合でも，SDIOカード・プロトタイプがSD規格に合っていれば，不具合は起きないはずである．万一，このメッセージが現れないときは，初期化シーケンスに失敗していることになるが，失敗したときのログと，SD-IDEを使って成功したときのログを見くらべれば，失敗の原因を特定できる．ログは，Optionsをクリックし，Log Windowにマウスを当てると出てくるプルダウン・メニューの中にあるOpenをクリックすると(図8)，ログのウィンドウが開き(図9)，どのようなコマンドが発行され，それに対してどのようなレスポンスがあったかを確認することができる．

ログのウィンドウでは，CMD52またはCMD53について，より詳しい情報を確認できる．確認したいコマンドをクリックして，左下のCMD Detailedボタンをクリックすると，Read

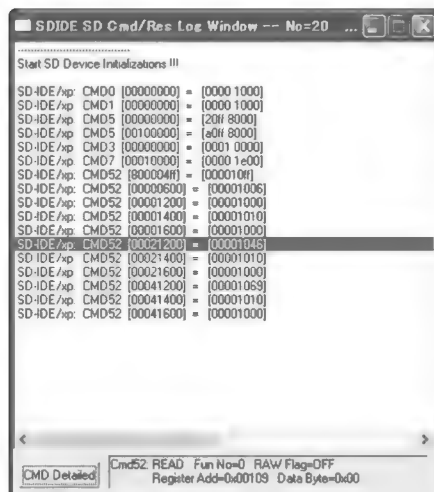


図10
CMD52やCMD53の詳しい情報を表示



図12
SDIOカードが抜かれたことを示すメッセージ

/Writeの種別や，ファンクション番号，アドレス，データなどの情報を確認できる(図10)．

ログをセーブするには，SD-IDEウィンドウのOptionsをクリックし，Log Windowのプルダウン・メニューのSaveをクリックする．しかし，その前にOptionsのSetup Log Fileをクリックして，ログ・ファイル・セットアップ・ダイアログ(図11)で，どのファイルにSaveするかを指定しておこう．

ログを見ると，通常の初期化シーケンス(CMD7まで)以外にもコマンドを発行しているが，順にI/O Enableレジスタの設定，I/O Readyレジスタの確認，各ファンクションのCIS情報が格納されているアドレスの確認を自動的に行っている．

SDIOカードは，いつでもSDIOスロットから抜くことができる．先ほど挿入したCG100-EDKをSDIOスロットから抜くと，直ちにSD-IDEのウィンドウにSDIOカードが抜かれたことを示すメッセージが表示される(図12)．

SDIOを閉じるには，Fileをクリックし，プルダウン・メニューの中のExitをクリックする．

ホストの設定

SDバスでは，1ビットのデータ・ラインを使う場合と4ビットのデータ・ラインを使う場合を選択できる．また，SDバスの周波数は0から25MHzの間の周波数を選択できるが，SD-IDEでは，SD Control Panelをクリックし，プルダウン・メニューのHostをクリックすると，ホスト・コントロール・ダイアログ(図13)が現れ，これらを選択できるようになっている．SDバスのデータ・ライン幅や周波数は，いつでも変更することができる．

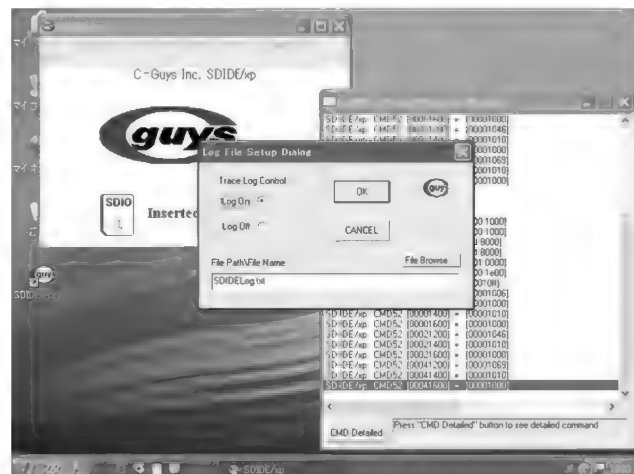


図11　ログ・ファイル・セットアップ・ダイアログ

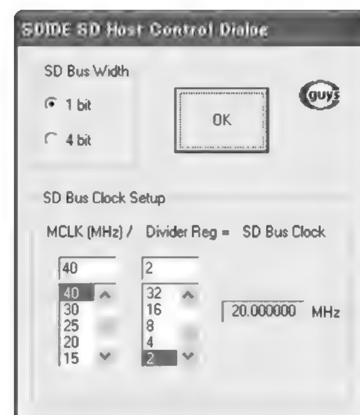


図13
ホスト・コントロール・ダイアログ

SDIOカード・プロトタイプのレジスタ・アクセス

SD Control Panelをクリックし，今度はCIA Accessにマウスを当てると出てくるプルダウン・メニューの中のCCCRをクリックすると，CCCR(カード・コモン・コントロール・レジスタ)ダイアログ(図14)が現れる．右端に各レジスタのリストが表示されるので，それをクリックして選択すると，そのレジスタの説明が左下のDescriptions:に表示される(図15)．レジスタの名前で指定できるので，仕様書を見ながらアドレス値を探す必要はない．小さなことだが，デバッグ効率を上げ，デバッグに対する心理的な障壁を取り除くには重要な要素である．

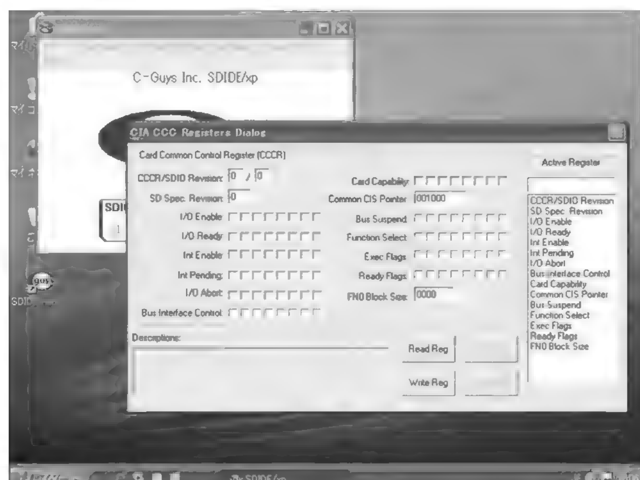


図14
CCCR(カード・コモン・コントロール・レジスタ)ダイアログ

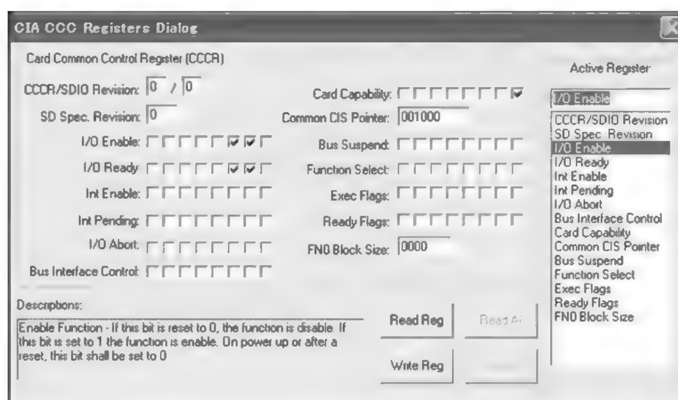


図15
レジスタのリストをクリックするとDescriptionに説明が表示される

SD Control PanelのCIA Accessにマウスを当て，今度はFBRをクリックすると，FBR(ファンクション・ベーシック・レジスタ)ダイアログ(図16)が現れる．ほとんどCCCRダイアログと同じような画面レイアウト，使いかたである．

CCCRもFBRもファンクション0に割り当てられているレジスタであり，すべてのSDIOカードに共通のレジスタである．ファンクション0には，これらのレジスタのほかにも，CIS(Card Information Structure)が割り当てられている．CISは，PCMCIAで設計されたもので，製造会社やプロダクトIDなどのカード情報を格納している．

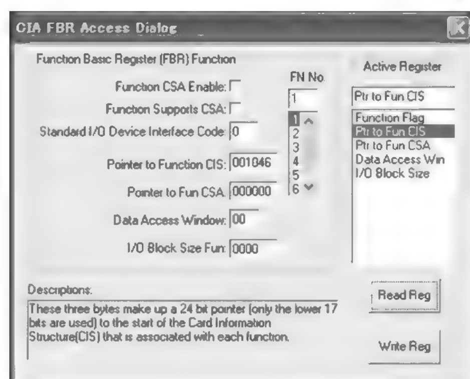


図16　FBR(ファンクション・ベーシック・レジスタ)ダイアログ

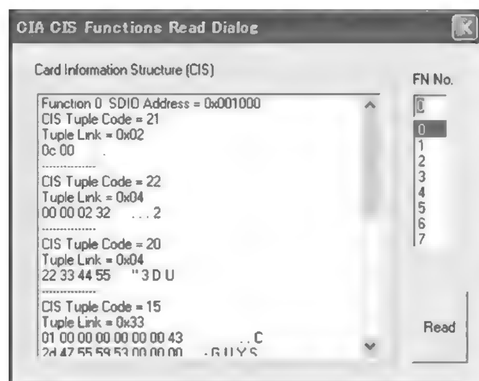


図17　CISファンクションの内容を読みたいときの動作

SDIOカード・プロトタイプのCISの確認

SDIOカードの初期化シーケンスを終えると，通常，SDIOホスト・ドライバはCISを読み込んでカードの属性を調べ，それに応じたSDIOクライアント・ドライバをロード，リンクする．CISの情報が読めない，あるいは情報の内容がまちがっていると，SDIOクライアント・ドライバの開発やテストを始められないので，SD-IDEによって確認しておく．

SD-IDEのウィンドウにあるSD Control PanelのCIA Accessにマウスを動かし，CIS Functionsをクリックすると，CIS読み込みダイアログが現れる．CISは，各ファンクションごとに用意されているので，ダイアログの右のほうにあるファンクション番号リストから読みたいファンクションの番号をクリックし，Readボタンをクリックすると，指定したファンクション番号のCISが読み出されて，内容が表示される(図17)．

CISは情報の種類ごとにタプルという情報単位で管理されていて，そのタプルのリンクド・リストで構成されているが，CIS読み込みダイアログでは，タプルごとにその内容を表示する．さらに，16進の数値だけではなく，それのASCII文字も同時に表示することで，内容確認作業を容易にしている．

実使用の場面では，SDIOカードが挿入，あるいは電源がON

Read Regボタンをクリックすると，直ちにSDコマンドが発行され，選択されたレジスタの値が読み込まれてそのレジスタのフィールドに反映される．また，Write Regボタンをクリックすると，ただちにSDコマンドが発行され，選択されたレジスタのフィールドに設定した値がSDIOカード・プロトタイプ内にあるレジスタに書き込まれる．

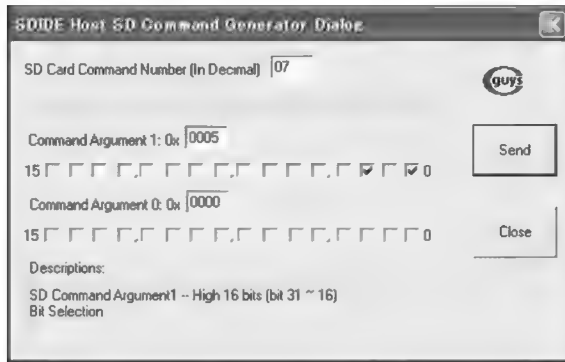


図18
一般的なコマンドを発行するためのダイアログを出すメニュー

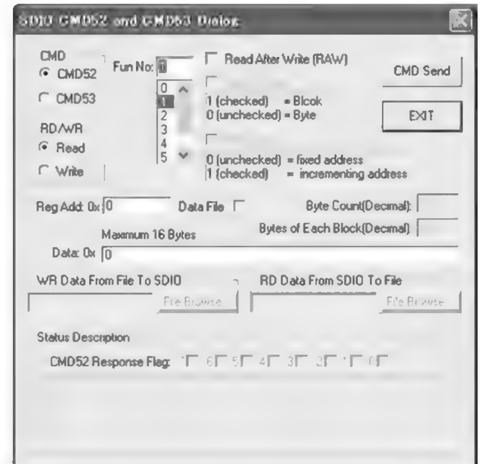


図19　CMD52とCMD53のダイアログ

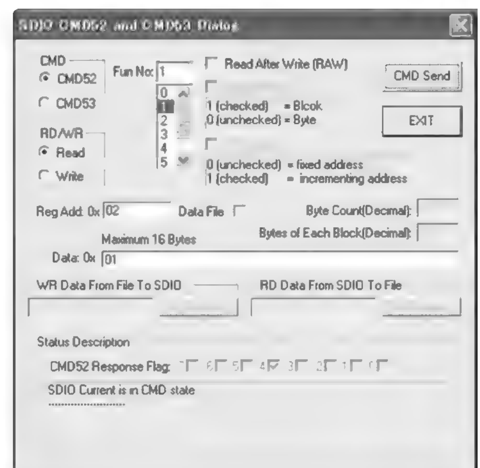


図20　読み込んだデータはデータ・フィールドに表示

されるたびに，初期化シーケンスに続いて，CISが読み込まれるが，SDIOカード・プロトタイプの動作確認においては，初期化シーケンスは必須であるものの，CISを毎回読む必要はない．また，CG100-EDKのROMには，あらかじめダミーのCISが書かれているので，SDIOカード・プロトタイプの動作試験段階ではCISを用意する必要もない．CIS読み込みコマンドに応答するのはSDIOカード・コントローラ側(ここではCG100)の仕事なので，設計者はCISの内容にだけ留意すればよい．

設計しているSDIOカードのCISを作ったら，それをCG100-EDKのROMに書き込み，SD-IDEのCIS読み込みダイアログを使ってタプルのリンク情報やタプルの内容を確認しておこう．

ファンクションの動作確認

それでは，各SDIOカード特有の動作確認に移る．前述したように，ファンクション0に割り当てられているレジスタのマッピングは，すべてのSDIOカードで共通のものだが，ファンクション1からファンクション7のレジスタ・マッピングは，それぞれのI/O機能に特有のものである．

ただし，BluetoothやGPSなど，レジスタ・マップがSDアソシエーションで規格化されているI/O機能も一部あるので，そのようなSDIOカードを設計する場合は，レジスタ・マップやI/O機能を規格に合うように設計したほうが良いのはいうまでもない．

さて，SDIOカード・プロトタイプのファンクション1やファンクション2に割り当てられているレジスタへのアクセスは，一般のコマンド発行機能を使用する．

SD-IDEのウィンドウのSD Commandをクリックしてプルダウン・メニューを出し，SendCmd5253をクリックする．プルダウン・メニューにはGeneratorもあるが，これはより一般的なコマンドを発行するためのダイアログを出すメニューである(図18)．一般コマンド発行ダイアログからは，任意のコマンドIDおよび任意のコマンド引き数を設定して，そのコマンドを発行できる．しかし，SDIOカードの場合，各ファンクションのレジスタをアクセスするのはCMD52またはCMD53だけなので，動作試験においては，Generatorをクリックする機会はあまりないだろう．

SendCmd5253をクリックすると，CMD52とCMD53のダイアログ(図19)が現れる．これはGeneratorの一般的なコマンド発行機能とは異なり，コマンド引き数を構成する各フィールドごとに値を設定できるようになっているので，いちいちコマンド引き数を組み立てる必要がなく，便利である．各フィールドの値を設定して右上にあるCMD Sendボタンをクリックすると，すぐに設定したコマンドが発行される．読み込みコマンドの場合には，読まれたデータはデータ・フィールドに表示される(図20)．

前回(2月号掲載)の記事で紹介した『SDIOインジケータカード』なら，CG100-EDKだけで実現できる．CMD52とCMD53のダイアログで，CMDをCMD52に設定し，RD/WRをReadに設定し，Fun Noを2(1でもよい)に設定し，Reg Addを

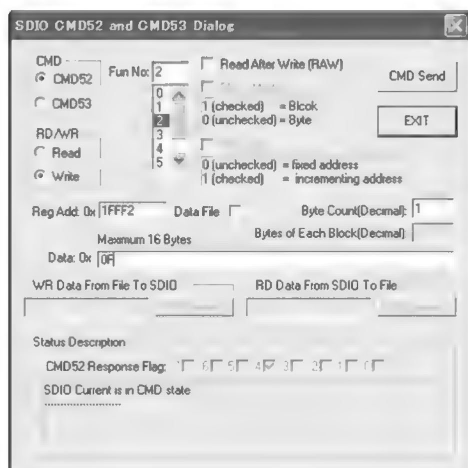


図21　SDIOインジケータ・カードのLEDを点灯させるCMD52とCMD53のダイアログ



図22　コマンド・リスト・ファイルの選択を促すダイアログ

1FFF2hに設定し，Dataに0Fhを設定して，CMD Sendボタンをクリックする（図21）．すると，SDIOインジケータ・カードのすべてのLEDが点灯するはずである（写真3）．

また，Dataフィールドの値を05hに設定してCMD Sendボタンをクリックすると，今度はGPIO1のLEDのGPIO1とGPIO3が消灯する．

Dataフィールドの値をいろいろ変えてCMD Sendボタンをクリックすれば，SDIOインジケータ・カードの動作試験を行える．ほかのSDIOカード・プロトタイプについても，同じように動作試験可能であるが，ここで示したような単純なものばかりではない．そのような場合は，あらかじめ発行したいコマンドのリストを作成し，SD-IDEのウィンドウのSD Control Panelをクリックして，出てくるプルダウン・メニューの中のOpen CMD Scriptsをクリックする．すると，コマンド・リスト・ファイルの選択を促すダイアログ（図22）が現れるので，作成したコマンド・リスト・ファイルを選択すれば，そのコマンド・リストに従って連続的にコマンドが発行される．

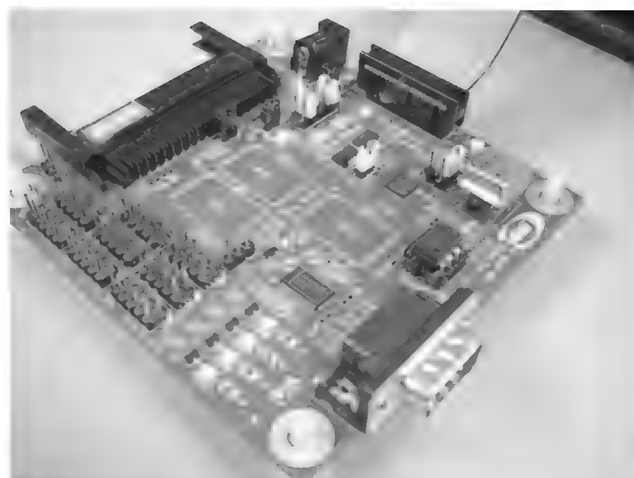


写真3　SDIOインジケータ・カードのLEDが点灯する

リスト1「SDIOほたるカード」の動作試験用コマンド・リスト

```
C-GUYS-CGSS-1.0
UINT8   WrtCommand=0x57
#   0x57 = W
UINT8   WrData=0x3F
#       0x1F = all 0
#       0x3F = all 1
#       0x5F = all 2
UINT8   One=0x01
UINT8   Zero=0x00
UINT8   RdFifo=0x0C
UINT32  CntCnt=0x0000
UINT32  RdCnt=0x0000
UINT32  MaxCnt=0x0100
UINT32 Result
UINT32 Temp
UINT32 TtlSize=0x01
UINT8 VAR0X83=0x83
UINT8 VAR0X24=0x24
UINT8 VAR0X00=0x00
UINT8 VAR0X03=0x03
#       CMD53W FUNC=1 OP=FIXED MODE=BYTE
#       SIZE=1 ADDR=0x13 VAR=WrtCommand
#       UART INIT
Cmd52 RW=W FUNC=1 RAW=OFF ADDR=3 VAR=VAR0X83
Cmd52 RW=W FUNC=1 RAW=OFF ADDR=0 VAR=VAR0X24
Cmd52 RW=W FUNC=1 RAW=OFF ADDR=1 VAR=VAR0X00
Cmd52 RW=W FUNC=1 RAW=OFF ADDR=3 VAR=VAR0X03
#       WRITE COMMNAD "W"
:infinity
CMD53W FUNC=1 OP=FIXED MODE=BYTE SIZE=1 ADDR=0x13 VAR=WrtCommand
:TOPLOOP
CMD53W FUNC=1 OP=FIXED MODE=BYTE SIZE=1 ADDR=0x13 VAR=WrData
BINARY_EXP CntCnt = CntCnt + One
BINARY_EXP Result = CntCnt < MaxCnt
IF TRUE GOTO TOPLOOP ELSE GOTO TOPEND
:TOPEND
CMD53R FUNC=1 OP=FIXED MODE=BYTE SIZE=257 TOTALSIZE=TtlSize
                                             ADDR=0x12 VAR=Temp
END
```

『SDIOほたるカード』の動作試験をするために作ったコマンド・リストをリスト1に示す．Open CMD Scriptsをクリックして，このコマンド・リストを選択すると，シンプル・スクリプト実行ダイアログ（図23）が現れる．このシンプル・スクリプト実行ダイアログでは，Stepボタンをクリックすると，ハイライトされている行だけが実行され，ハイライトが次のステッ

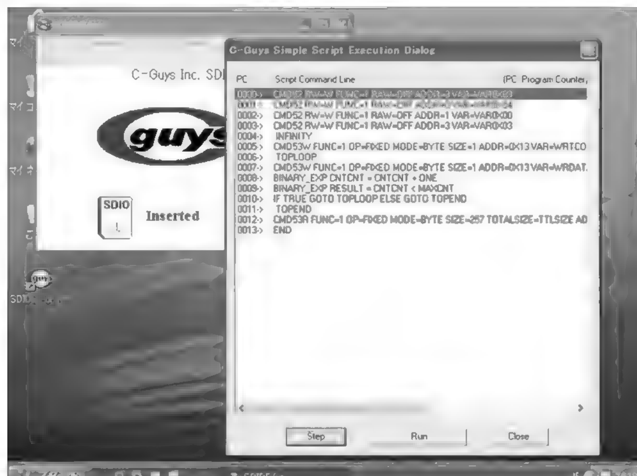
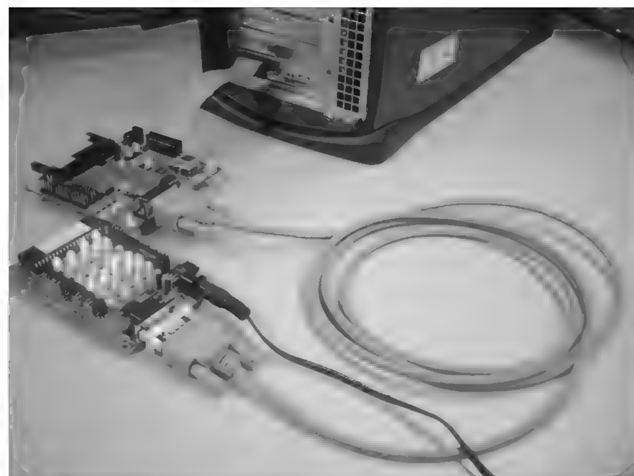


図23　シンプル・スクリプト実行ダイアログ


写真4「SDIOほたるカード」の動作試験

プに移動する．

また，Runボタンを押すと，ハイライトされている行から連続的に実行される．実行中は，そのときに実行されている行にハイライトが移動する．また，RunボタンがStopボタンの表示に変わり，いつでもStopボタンをクリックして実行を中止できる．実行開始の行を変えたい場合は，実行を開始したい行をクリックするだけで，その行がハイライトされ，その行から実行できるようになる．

リスト1のコマンド・リストを先頭から実行させると，発光パターンを制御するデータがファンクション1に割り当てられているUARTを通してLEDを制御しているプロセッサに伝えられ，LEDの点滅パターンが変化する(写真4)．

コマンド・リストは，単純なコマンドの並びだけではなく，一種のスクリプト言語のようになっていて，変数への代入/参照やプリント・アウトの指示，条件分岐などが使える．詳しくは，SD-IDEのマニュアルを参照してほしい．

まとめ

以上，SDIOカードのドライバ・ソフトウェア開発について概説し，各種OSの対応状況にも触れた．また，SD-IDEを使ってどのようにSDIOカード・プロトタイプの動作試験を行うかを示唆し，同時にカードが挿入されてからのSDホストの動きを解説した．

さて，いよいよ次回で「SDIOカード開発入門」の連載も最後を迎える．SDIOカードが普及することを願いつつ，最終回では，SDIOカードの将来と展望について述べる．

■SDIOカード・コントローラ・チップCG100およびSDIO開発環境ツールSD-IDEの問い合わせ先
シイガイズ(株)
E-mail：Sales@c-guys.com

やそしま・ひろゆき　シイガイズ(株)技術開発部　Senior SOC Engineer



Appendix

SDIO *Now!*プログラムについて

中山 宏之

SDIO *Now!*プログラム(BSquare社)に加入することにより，Windows CEデバイスやSDIOホスト・コントローラなどの製造者，SDIOカードの開発をしている会社は，それぞれにSDIO対応のメリットを受けることができる．

Windows CEでSDIOをサポートするための問題点

マイクロソフトの組み込み向けオペレーティング・システムWindows CEは，組み込み用開発ツール「Platform Builder」を利用することによって，あらゆる組み込み機器向けにWindows CEをカスタマイズして搭載することができる．そのため，(原理的には)かなり大規模な機能拡張や，場合によっては不要な機能の削除も可能になっている．

ある機能がある特定のWindows CEハードウェアだけに搭載されているような場合，デバイス開発者は特定ハードウェアだけの独自の拡張をデバイス・ドライバで実現してWindows CEに搭載する．

ところが，SDIOのような相互運用を考える必要のある規格の場合は(マイクロソフトを除くと)1社だけでそのすべてをサポートすることは非常に難しい．SDIOは，まさにこの分野の応用アプリケーションである．

SDIOへの対応としては，以下の三者の立場が考えられる．

- ホスト・コントローラの開発者

SDIOホスト・コントローラやそのIP[注1]を開発している会社は，それをいろいろな機器ベンダに販売したい．そのため対応OSや対応SDIOカードをできるだけ多くサポートしたい．ところがこのような会社は一般にハードウェアは得意なのだが，このようなソフトウェア・サポートは荷が重い．

- SDIOカードの開発者

SDIOカードを開発している会社は，携帯機器を使用する一般消費者や機器メーカへのOEM販売などを行う．そのためには対応する機器をできるだけ増やしたい．こちらも少なくともOSごとに，場合によっては一つ一つの機種ごとに対応ソフトウェアを用意する必要がある．

- 携帯デバイスの開発者

携帯デバイスの開発者は機器の魅力を高めるため，SDIOの採用を考える．ところが，どのSDIOホスト・コントローラを採用し，どのSDIOカードの動作保証をするかで悩むことになる．できるだけ労力をかけずにSDIOサポートを実現したい．

SDIOの実装上の問題は，実は上記三者の立場の違い以外にもある．それは(たとえばWindows CE上において)SDIOを扱うためのソフトウェア標準が(現在までのところ)存在しないということである．SDカード・アソシエーションの規定しているSDIOの規格は，おもにハードウェア実装上のプロトコルの規定となっており，それが特定OS上でどのように実装されるかというところまでは規定されていない．したがって，同じOS上でも互換性のないドライバというのが発生する可能性がある．

SDIO Now!プログラムとは[注2]

SDIO *Now!*プログラムは，現状のWindows CE OS上でSDIOを実現するための標準ソフトウェアを提案するものである．その大まかな構成は図Aのようになる．

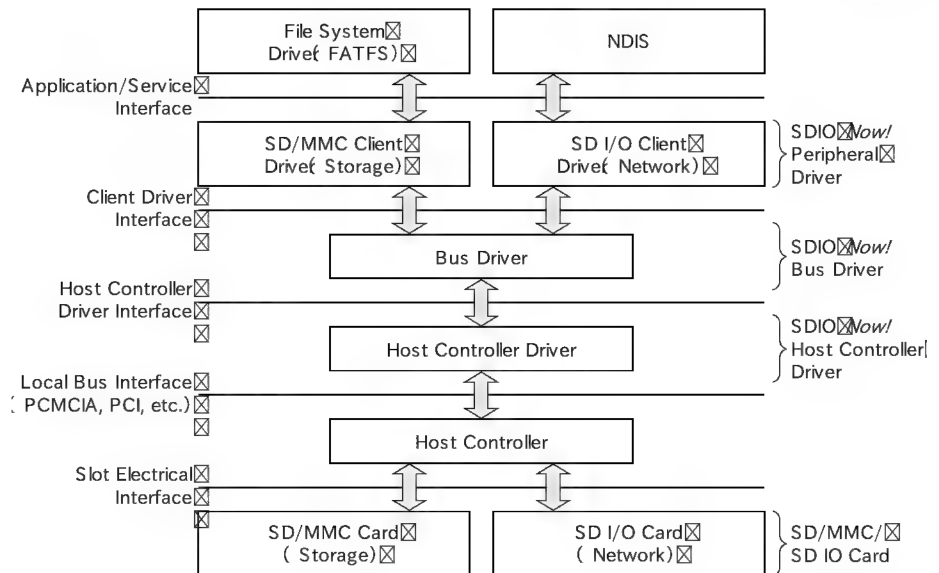
この図で中心になるのがBSquare社で開発したSDIO *Now!* Bus Driverである．これはWindows CE.NETやPocket PC2003でOS標準のバス・ドライバ[注3]として動作するモジュールであり，下側にはHost Controller Driver Interfaceを，上側ではClient Driver Interfaceをサポートしている．ホスト・コントローラ・ベンダはHost Controller Driver Interfaceに対応したドライバを記述することでSDIO Now!対応のすべてのSDIOカードを使用することができる．一方，SDIOカード開発者は，Client Driver Interfaceに対応したドライバを開発することで，SDIO *Now!*対応のすべてのSDIOホスト・コントローラやSDIO *Now!*対応Windows CE機器で同じクライアント・ドライバを使用することができる．

SDIO Now!プログラムには，上で述べた3種類の立場ごとにプログラムが用意されており，それぞれの立場でのメリットを得られるように考えられている[注4]．

- SDIO *Now!* for Semiconductor Vendors

SDIO *Now!*バス・ドライバ(ソースまたはバイナリ)，SDメモリ/MMCストレージ・クラス・バイナリ・ドライバ，Host Driver DDK，サンプルSDIO Host Driverソース・コード(XScale向け)が提供される．

注1：IP＝知的所有権であるが，ここではSDIOホスト・コントローラを実現するためのマクロセルなどを指している．
注2：SDIO *Now!*オフィシャルWebページ　`http://www.bsquare.com/products/sdionow/default.asp`
注3：Windows CE 3.0やPocketPC 2000/2002では標準バス・ドライバのしくみは存在しないが，Windows CE 3.0用にビルドされたSDIOバス・ドライバの働きにより，CE.NET用と同じホスト/クライアント・ドライバを使用することができる．
注4：プログラムに加入するためには費用が必要．詳細はビースクウエア営業担当窓口まで問い合わせのこと．



図A　SDIO *Now!*ドライバの構成

● SDIO *Now!* for Peripheral Manufacturers

SDIO *Now!*バス・バイナリ・ドライバ，SDメモリ/MMCストレージ・クラス・バイナリ・ドライバ，Client Driver DDK，およびサンプルSDIO Client Driverソース・コード(Block，UART，NDIS，Bluetooth)が提供される．

● SDIO *Now!* for Device OEMs and ODMs

SDIO *Now!*バス・ドライバ(ソース)，Host Driver DDK，Client Driver DDK，サンプルSDIO Host Driverソース・コード(XScale向け)，サンプルSDIO Client Driverソース・コード(Block，UART，NDIS，Bluetooth)が提供される．

また，SDIO *Now!*メンバ各社はビースクウエアのサポートWebサイトにアクセスする権利が与えられ，ここを通じてビースクウエアはバス・ドライバやDDKの最新バージョンを提供する．また，ドライバ配布を希望するホスト・コントローラ・ベンダやSDIOカード・ベンダは，ここを通して機器ベンダにドライバを配布することができる．SDIO対応機器ベンダは，ダウンロードしたドライバを(ライセンス条項にしたがって)自分の機器に組み込んだ状態で出荷することができる．

将来との互換性を確保するには

SDIO *Now!*プログラムは，一種のアーリ・アクセス・プログラムである．その意味は，もし顧客の役に立つものであるなら，規格が完全に安定する前でも使用可能なものを提供しようということである．逆にいえば，将来的には何らかの規格変更を受けるかもしれない．

たとえば，マイクロソフトは次のWindows CEバージョン「Macallan」でSDIOのサポートを予定している．したがって，BSquare社独自のプログラムであるSDIO *Now!*プログラムに加入しなくても，時が経てばWindows CE自体にSDIOサポート機能が搭載され，同時にある程度のホスト・コントローラやSDIOカードもサポートされると思われる．また，SDIO *Now!*プログラムも，実際には期間限定のプログラムであり，マイクロソフトからSDIO対応のWindows CEがリリースされた時点で終了することになっている．

このような事情にもかかわらず，現在SDIO *Now!*プログラムを推進する理由は，

● 現状のOSですぐにSDIOを利用したいユーザをサポートする

たとえば，将来的にCEでSDIOがサポートされても，そのドライバをWindows CE 3.0やCE.NET 4.xなどの旧OSにもってくることはできない．

● Macallanのリリース日が未定

現時点で2004年のいつか，と考えられているが，まだ決定していない．またMacallanベースのPocket PCとなるとさらに先になる?

● Macallanでのホスト・コントローラ・サポート/デバイス・サポートが未定

現状でも複数のホスト・コントローラが存在するが，そのすべてがサポートされるわけではないと考えられている．

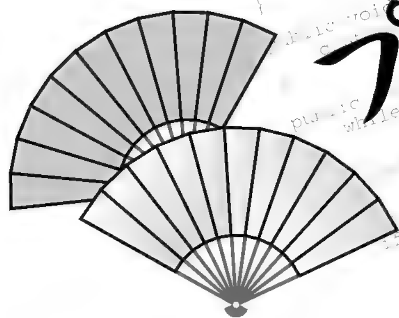
などである．要するに今，SDIOをサポートすることに価値を見出せる会社がSDIO *Now!*プログラムに加入しているのである[注5]．

とはいえ，BSquare社はマイクロソフトとも緊密に連絡し，マイクロソフトのMacallanでのSDIO開発をサポートしている．願わくば，SDIO *Now!*向けに開発したホスト・コントローラ・ドライバやクライアント・ドライバが最小限の修正で次期Windows CEで利用可能であることを期待している．

■ SDIO *Now!*プログラムの問い合わせ先
BSquare Tiwan
`tonychiang@bsquare.com`(National Account Manager)

なかやま・ひろゆき

注5：現在SDIO Now!に加入している会社のロゴが `http://www.bsquare.com/products/sdionow/licensees.asp` で紹介されている．





プログラミングの要

宮坂 電人

第10回

データ構造とアルゴリズム

アルゴリズムをまとめる意義

前回紹介した「Mastering Algorithms with C」注1の続きです．プログラミングを行う場合，まったくの新人ならともかく，ある程度の経験を積むと，たいていは以前にもこれをやったことがある，という場面に必ず遭遇します．データ構造，アルゴリズム，デザイン・パターンという用語で説明されているものは，多くのプログラマが頻繁に遭遇するであろうパターンを抽象化してまとめたものです．

しかし，何のためにまとめるのでしょうか．よほどの天才プログラマならともかく，一般的な知能と行動パターンをもつプログラマが，いきなり素手で未知の課題に挑戦しても成功したためしはありません．未知の課題に関する知識がないのが最大の理由ですが，そのほかにも課題に適したプログラミング戦略や戦術を知らないからという理由もあります．事務プログラムに何年も携わっているにもかかわらず，ゲーム・プログラムができない人は珍しくありません．高度な数学的処理が得意でもP2Pを使ったファイル転送の方法を知らない人もたくさんいます．しかし，与えられた課題に対する適切なプログラミング戦略や戦術を教えることで，ある程度はカバーできるはずです．

データ構造の利点

「データ構造」というと，何かしら難しいものを想像してしまいます．中には，簡単なものをわざわざ難しくすることで，かえって不利な目に遭うのではないかという指摘や批判があります．たしかに，世の中には簡単にすむことを，わざわざめんどうに高級ぶって，しんどいことを行っている例は多々あります．

しかし，簡単なままにしておくと，かえってしんどい目に遭うからこそ，ある程度の構造や体系を取らせることもあります．むしろ，そちらが最大の理由でしょう．

たとえば，単純な配列を使うと型をむりやりキャストしないといけない，あるいは固定長だから融通が利かないというときに，構造体にしたり可変長のフレキシブルなデータ構造を採用することはありがちです．単純な配列で十分なら構造体やフレキシブルなデータ構造にする意味はありませんし，むしろややこしい方向で疲れてしまうだけです．「Mastering Algorithms with C」では，データ構造を採用するメリットとして以下のようなものがあると指摘しています．

● **効率性（Efficiency）**

データの登録や検索などで有利になります．多数のデータを検索する場合，単純な配列では配列の最後に記録されたものは最後にヒットするため，配列のサイズが大きくなればなるほど不利になります．ところが，ハッシュ・テーブルあるいはバイナリ・ツリーを利用することで検索する速度を向上させることができます．

単純な実装では，規模が小さいうちはよいのですが，規模が大きくなるにつれ，だんだん不具合が生じ，そのうち取り返しのつかない事態に追い込まれることもありえます．

● **抽象化（Abstraction）**

具体的な実装方法を気にしないことで，汎用的にデータ構造の使い回しがしやすくなります．

たとえば，スタックやキューという構造を考えてみましょう．単純に固定長配列にデータを登録して取り出す方法では，その配列の有効サイズはいくらかとか，配列が足りなくなった場合にどうするかといった余計な心配があります．また，メモリが足りない場合にどうするかという心配もあります．中身をどう実装しているかは大事ですが，どのような実装方法を採用しても，それぞれの実装方法にはそれぞれの特徴や限界があります．

そこで思い切った考えかもしれませんが，具体的な実装，ここでは固定長配列なのか，あるいは連結リストで実装しているのか，メモリ不足のときの対応をどうするかなどはいったん忘れてしまい，単に「なにものか」に記録するという考えに徹します．そして「なにものか」に対するサービスとして必要そうなものを列挙していき，そこから次のステップへ考えを進めていきます．

よく考えると，これは初歩的なオブジェクト指向の考え注2ですが，こうした考えは高度なものではなく，比較的規模の小さ

注1：http://www.oreilly.com/catalog/masteralgoc/を参照．
注2：厳密にいえば「抽象データ型」という考え．

い段階でも導入することができます．また，小さい段階で導入することで将来的に規模が大きくなったときに移行しやすくなる考えでもあります．

● **再利用性（Reusability）**

汎用的に使い回せるようにしたデータ構造は，当然のことながら再利用性が高くなります．このことにより工程が縮むことはもちろんのこと，実績のあるデータ構造はテストが繰り返されているため，バグが少なく，信頼性や堅牢性が高くなります．

アルゴリズムの利点

データ構造とも共通しますが，アルゴリズムの利用や記録にはさまざまなメリットがあります．「Mastering Algorithms with C」では，以下のようなメリットがあると指摘しています．

● **効率性（Efficiency）**

プログラムは，最終目的が何であれ，途中で遭遇する命題にはいくつか共通するものがあります．たとえば，検索やソートという命題は数多く遭遇するものであり，それゆえたくさんの人たちの挑戦を受け，検討されています．それらを調べることで，より効率のよい解決方法を取ることができます．

● **抽象化（Abstraction）**

具体的な実装方法はどうであれ，よく遭遇する命題の解決方法（すなわちアルゴリズム）は似通ったものがあります．それらを調べることで，まったくの1からではなく最良の解決方法を実装することができます．

また，解決方法は，特定のプログラミング言語や詳細な実装をなるべく排除して抽象化することで汎用的に解決する手段として検討する余地が出やすくなります．

● **再利用性（Reusability）**

アルゴリズムは，さまざまな命題の解決策として使い回しが利きます．このことにより工程が縮むことはもちろんのこと，実績のあるアルゴリズムはテストが繰り返されているため，バグが少なく信頼性や堅牢性が高くなります．

アルゴリズム設計のアプローチ

アルゴリズムを構築する場合，いくつかのアプローチが考えられます．天才的な頭脳の持ち主なら「突然ひらめいた」とか「何者かの啓示を受けた」と説明することもあるでしょうが，たいていは泥臭いアプローチの末にようやく見つけるものです．「Mastering Algorithms with C」では，以下のようなアプローチがあると指摘しています[注3]．また，一つだけのアプローチではなく，複数のアプローチを組み合わせてアルゴリズムが構築されることもあります．

● **ランダムなアルゴリズム（Randomized algorithms）**

文字どおり，ランダムな決定によって問題解決をはかるアプローチです．「ランダム」といういいかたは誤解をまねきやすいのですが，乱数といえども統計的に現れる特性（均等にまんべんなく出現する特性など）に着目すれば，へたにマジメに考えこむアプローチよりも，かえって問題を手早く解けることがあります．

● **分割統治アルゴリズム**
（Divide-and-conquer algorithms）

複雑で巨大と思える問題を小さく分けて，各個撃破して解決するアプローチです．その際，

1）分割する（divide）
2）征服する（conquer）
3）結合する（combine）

の3段階に分けて解決を図ります．

● **ダイナミック・プログラミングによる解決**
（Dynamic-programming solutions）

「動的計画法」という訳もあります．分割統治と同様，小問題に分割して各個撃破するアプローチですが，小問題の解をあらかじめ「表」にして保存しておきます．さらに，小問題の解を組み合わせ，より大きな問題の解の作成を行おうというアプローチです．分割統治がトップダウン式なのに対し，ダイナミック・プログラミングのほうは小さい問題の解をボトムアップ式に積み重ねて大きい問題を解こうという逆方向のアプローチです．

● **貪欲なアルゴリズム（Greedy algorithms）**

その時点において最良と判断したものを選択するアプローチです．ただし，このアプローチは局所的には最良の判断であっても，全体的にも最良だという保証がないという問題点があります．

● **近似アルゴリズム（Approximation algorithms）**

完璧な最良を求めるのではなく，このくらいならまあ良い（good enough）という近似な良を選択するアプローチです．このアプローチの例として，「巡回セールスマン問題」がもっとも有名でしょう．

再帰を使う場合，使わない場合

再帰はシンプルな考えでありながら，どういうわけか難解であると誤解されることがあります．そうなってしまう理由として，たいていのプログラマは手続き指向的なプログラミング方法やプログラミング言語に慣らされているため，再帰を演習する機会がほとんどなかったり，再帰を使わなくても解決できる状況が圧倒的に多いと考えるからでしょう[注4]．

また，再帰の例を説明するために，学校を卒業すると，ほとんど使う機会のない階乗計算のような考えで説明されることも，

注3：実際には，この本で指摘した以外のアプローチもあることに注意．
注4：「多いから」ではなく「多いと考えるから」と書いている点に注意してほしい．手続き指向に最適になるよう思考に枠がはめられているからである．

リスト 1　for文による繰り返しの例

```
static void test1()
{
    int i;

    for(i = 1; i <= 5; i++){
        printf("%d¥n",i);
    }
}
```

リスト 2　再帰による繰り返しの例

```
static void test2sub(int i,int j)
{
    if(i <= j){
        printf("%d¥n",i);
        test2sub(i + 1,j);
    }
}

static void test2()
{
    test2sub(1,5);
}
```

リスト 3　機械語レベルで見たtest2subとtest2

```
_test2sub   proc   near
    push    ebp
    mov     ebp,esp
    push    ebx
    push    esi
    mov     esi,dword ptr [ebp+12]
                        ;int j はesiレジスタに代入される
    mov     ebx,dword ptr [ebp+8]
                        ;int i はebxレジスタに代入される

    cmp     esi,ebx     ;jとiを比較する
    jl      short @7    ;(j < i)ならこのルーチンを抜ける

    push    ebx         ;i
    push    offset s@+4 ;"%d¥n"の文字列が格納されているアドレス
    call    _printf     ;printf("%d¥n",i);
    add     esp,8       ;スタック・レベルを戻す

    push    esi         ;j
    inc     ebx         ;i+1
    push    ebx
    call    _test2sub   ;test2sub(i+1,j);
    add     esp,8       ;スタック・レベルを戻す
@7:
    pop     esi
    pop     ebx
    pop     ebp
    ret
_test2sub   endp

_test2  proc   near
    push    5
    push    1
    call    _test2sub   ;test2sub(1,5);
    add     esp,8       ;スタック・レベルを戻す
    ret
_test2  endp
```

リスト 4　機械語レベルで見たtest1

```
_test1  proc   near
    push    ebx
    mov     ebx,1       ;i=1
@2:
    push    ebx         ;i
    push    offset s@   ;"%d¥n"の文字列が格納されているアドレス
    call    _printf     ;printf("%d¥n",i);
    add     esp,8       ;スタック・レベルを戻す
    inc     ebx         ;i++
    cmp     ebx,5
    jle     short @2    ;(i<=5)なら@2へジャンプ

    pop     ebx
    ret
_test1  endp
```

敬遠される原因になると筆者は考えます．しかし，よく観察すると，ツリー状にデータ構造を構築したり，マクロ構造の中にミクロ構造が相似して現れる状況は珍しくありません．たとえば，ファイルをディレクトリに格納する状況はまさにツリー状に構築している状況でしょう．

また，LISPなどの関数パラダイムのプログラミング言語では再帰は必須であり，たとえば繰り返し処理は再帰を使って表現します．同じことを手続き指向のプログラムでもできないわけではありません．

たとえば，C言語でリスト 1のように記述するプログラムがあるとします．これを再帰を使ってリスト 2のように記述することも可能です．とはいっても，実際にこんなふうに記述する手続き指向プログラマはあまりいません．というのもプログラマが再帰に慣れていないことはもちろんのこと，再帰を使うとスタックを消費したり関数呼び出しの回数が増えることによる弊害を警戒するからでしょう．

リスト 2で示したプログラムを機械語レベルに翻訳するとリスト 3のようになります注5．test2subを再帰的に呼ぶため，その前でスタックを8バイト消費するpush命令が2回，もちろんtest2subのコールもあります．

ちなみに，再帰を使わないtest1の場合はリスト 4のようになり，この結果だけを見れば再帰を使わないほうが有利です．ただし，いつでも再帰が手続き指向プログラミング言語で不利だとは限らず，状況によっては再帰を使うほうが有利な場合もあります．もっとも，この見極めは口でいうほど簡単ではなく，ある程度の経験を要するのかもしれません．

末尾再帰の展開のされ方

再帰の中でも，自分自身の呼び出しがルーチンの最後で行われる形態のものを「末尾再帰（tail recursion）」と呼ぶことがあります．たとえば，さきほどのtest2subは末尾再帰の一例です．末尾再帰になっているものはコンパイラの最適化によって，自分自身への呼び出しをコール文ではなくジャンプ文に展開されることがあります．その影響でスタックの消費を減らしたり無くすことができ，再帰ならではの不利をある程度は回避することができます．

たとえば，さきほどのtest2subとtest2は，もしも末尾再帰に着目した最適化が強力なコンパイラがあったと仮定すれば，リスト 5のように展開されるところでしょう注6．見かけ上

注5：この結果はBorland C++ 5.5.1 for Win32を使って検証したもの．

リスト 5　末尾再帰に着目した最適化

```
_test2sub   proc   near

    cmp     esi,ebx      ;jとiを比較する
    jl      short @7     ;(j < i)ならこのルーチンを抜ける

    push    ebx          ;i
    push    offset s@+4  ;"%d¥n"の文字列が格納されているアドレス
    call    _printf      ;printf("%d¥n",i);
    add     esp,8        ;スタック・レベルを戻す

    inc     ebx          ;i+1
    jmp     _test2sub    ;test2sub(i+1,j)
                         ; ↑コールではなくジャンプに展開する
@7:
    ret
_test2sub   endp

_test2  proc  near
    push    esi
    mov     esi,5
    push    ebx
    mov     ebx,1
    call    _test2sub    ;test2sub(1,5);
    pop     ebx
    pop     esi
    ret
_test2  endp
```

リスト 6　末尾再帰の見かけ上の展開

```
static void test2sub(int i,int j)
{
TOP:
    if(i <= j){
        printf("%d¥n",i);
        i = i + 1;
        goto TOP;
    }
}
```

リスト 7　疑似的なオブジェクト指向のヘッダ・ファイル

```
/* 疑似オブジェクトへのポインタ */
typedef struct PseudoObject *PObjectPtr;

/* 疑似オブジェクトの発生と初期化 */
PObjectPtr PseudoObject_new();

/* 疑似オブジェクトの終了化と解放 */
void PseudoObject_delete(PObjectPtr iObj);

/* 何らかのサービス */
void PseudoObject_service(PObjectPtr iObj);
```

は元のソースに対して，リスト 6のように先頭へgoto文が挿入されたかのように変形されるわけです．

データ構造の説明と実装

一般的なアルゴリズムの解説本ではデータ構造について説明するとき，その構成を見せることを重視するためか再利用性を考えない説明になっていることがあります．平気でグローバル変数を使っていたり，例題で説明している状況でしか通用しないような汎用性に欠ける記述になっているため，その説明だけを頼りに実装するのがめんどうであったり，毎回似たようなコードを記述する要領の悪い運用に陥ることがあります．

「Mastering Algorithms with C」がよくできていると評価すべきところは，説明で作成するデータ構造をあたかも抽象データ型[注7]として再利用しやすいように整理している点です．利用する側からはデータ構造の中身を意識させるのではなく，どういうサービスが用意されているかが興味の中心となるようにしています．

同書では，データ構造の中身を構造体として記述していますが，たとえばDataStructという型を用意したなら，

- DataStruct_init：初期化処理
- DataStruct_destroy：終了処理

の二つが用意されていて，サービスを利用する前に必ずDataStruct_initを呼び，利用が終わって二度と使わないならDataStruct_destroyを呼ぶという前提で，いろいろなサービスを使わせるようにしています．C++に慣れている人ならDataStruct_initはコンストラクタで，DataStruct_destroyはデストラクタに相当するものだと想像するところでしょう．C言語レベルでは，構造体をじょうずに利用することで疑似的なオブジェクト指向や抽象データ型を実現することが可能です．

たとえばリスト 7のように構造体の中身を明らかにせずに[注8]疑似オブジェクトとしてしまい，サービスを利用する側からはサービスの手続きしか使えないようにするくふうがあります．こうすることで内部実装を意識させないことはもちろん，仕様変更や仕様追加に際して構造体の中身を変更したり追加しても，サービスを利用する側に極力影響を与えないようにするくふうができます．

これこそ，まさしくオブジェクト指向のメリットとしてあげられている，実装の詳細を遮蔽(しゃへい)することで利用者側に悪影響を与えないくふうです．

連結リストのメリット

同じ型のデータを複数登録する際に配列はよく使われますが，同時に制約もあります．

1) 途中で格納サイズを変更できない[注9]
2) 途中への挿入や途中のデータの削除で大量転送が発生して

注6：厳密にいえばレジスタの破壊を防ぐための保存があったり，レジスタが破壊されてもよいかの検証などがあって，ここで示したような最適化がなされるかは疑問であるが．
注7：データの中身の構成，および，それらに対するアクセスやサービスを一体化したデータの型．
注8：「プログラミング言語C 第2版」のA8.3「構造体と共用体の宣言」で説明されている「不完全型」としての利用方法．
注9：ただし可変長にできるプログラミング言語やライブラリもある．

処理時間が長くなる
3) 検索を行うときに，その対象が最後に存在した場合，時間がかかる

ここで示した制約のうち，1)と2)を大幅に減らせるのが「連結リスト(linked list)」というデータ構造です．簡単にいえば，登録するデータに隣のデータへのポインタ(あるいは参照)を付加させた形態です(図1)．「Mastering Algorithms with C」では，以下の三つの形態を紹介しています．

- 単方向連結リスト(singly-linked lists)：一つのデータに隣へのポインタが一つ付いた連結リスト
- 双方向連結リスト(doubly-linked lists)：一つのデータに隣へのポインタが二つ付いた連結リスト
- 環状連結リスト(circular lists)：環状になっていて終端を作らない連結リスト

連結リストが格納サイズを簡単に変更できたり，途中への挿入/削除が短い処理時間ですむ秘訣は，まさしく隣へのポインタ(あるいは参照)があるおかげです．というのも，挿入/削除は単に連結リストの途中で隣へのポインタを書き換えるだけの手間ですし，それに伴って転送や移動がいっさい生じないからです(図2)．ただし，登録するデータとは別に隣へのポインタに必要なサイズが余分に必要になるので，配列よりも全体のサイズが増える傾向にあります．しかし，全体のサイズを取るか，処理時間を取るかはトレード・オフの問題であり，配列がよいか，それとも連結リストがよいかは状況によって変わってくるでしょう．

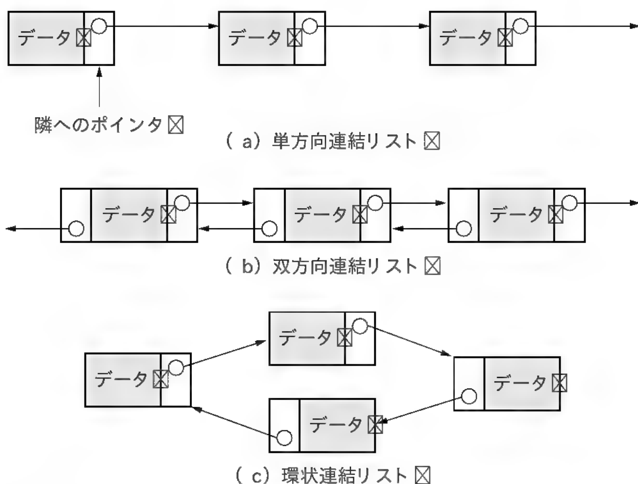
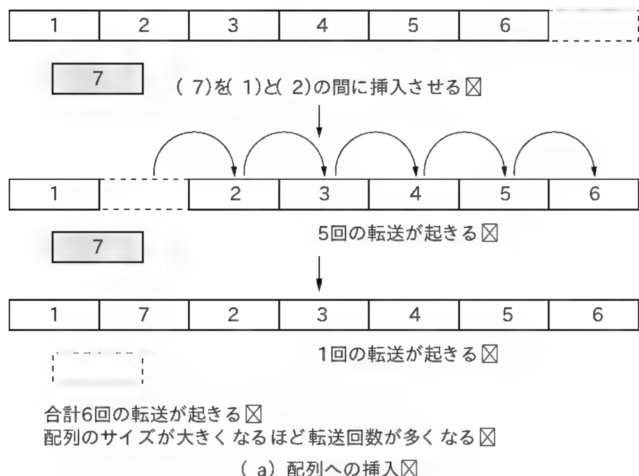


図1 連結リスト



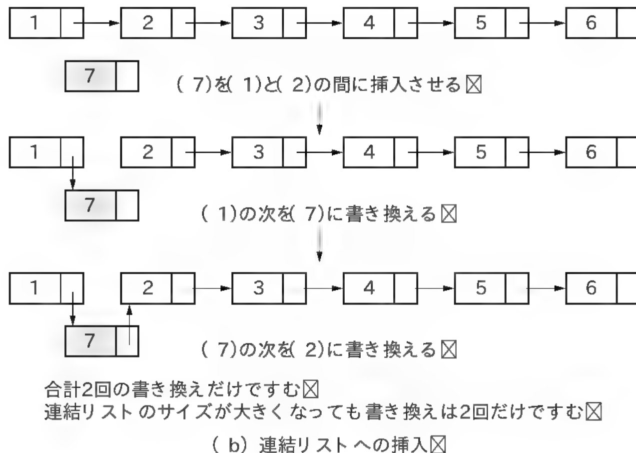


図2 配列と連結リストへの挿入

単方向連結リストの実装

単純に単方向連結リストを実装するなら，記録したいデータを構造体にし，隣へのポインタを追加すればよいのですが，そうすると汎用性が失われます．なぜなら記録したいデータは案件によって変化するからです．そのため，もうひとひねりして記録したいデータへの汎用ポインタ(あるいは参照)と隣へのポインタを構造体にします(図3)．Javaで実装する場合，構造体

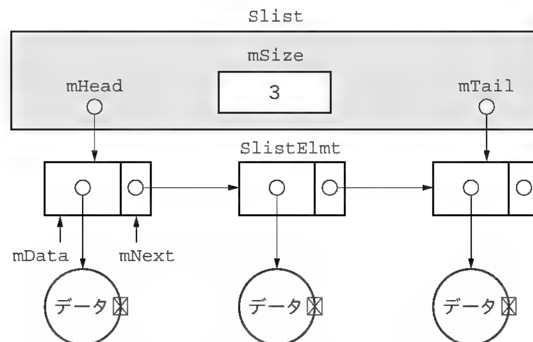


図3 SlistElmtとSlistを使った実装

はないので，それはクラスという形で実現します．記録したいデータへの参照（mDataと隣の要素への参照（mNext）は変数にしますが，これへのアクセスは専用メソッドを通して行わせます．具体的には，リスト8のように実装します．getDataメソッドがあるのにsetDataメソッドがないのは，SlistElmtと記録したいデータは一体化して取り扱うため，途中でmDataの書き換えをさせないという考えだからです．

SlistElmtは一つの記録であり，連結リスト全体を意味しません．全体はSlistという別のクラスで取り扱います．つまり，

- Slist：単方向連結リスト全体を取り扱うクラス
- SlistElmt：連結リストに記録する一つの要素を表現したクラス

という違いです．

ここで，単方向連結リストに必要なサービスを考えてみましょう．リストに記録している要素の個数（mSize）は必須ではありませんが，いくつの要素を記録しているかを手早く知るには必要です．これがないと，リストの先頭から要素をたどって個数を勘定しなければならず，当然のことながら処理速度が低下します．また，リストの最初に記録する要素（mHead）は必須です．これがないと，そもそも記録や取り出しができなくなります．リストの最後に記録する要素（mTail）がないと，リストの末尾に記録したいとき，先頭から要素をたどって末尾を調べることになり処理速度が劣化します．

以上より，フィールドはリスト9のようになります．また初期状態はいずれも0あるいはnullになるので，コンストラクタはリスト10のように実装されます．

さて，ここで単方向連結リストで必要になりそうなサービスを考えてみましょう．最初に必要になるのは，連結リストに対する登録です．リストの任意の場所に追加するメソッド（insertNext）を実装します（リスト11）．登録があるなら削除も必要です．リストの任意の場所から削除するメソッド（removeNext）を実装します（リスト12）．また，連結リストの登録数をえるメソッドなど，ほかのサービスも必要になるでしょう．これらはリスト13のように実装します．

Slistの利用例

Slistを使ったサンプルは，リスト14のようになります（図4）．先頭の要素を削除するのは簡単ですが，末尾の要素を削除するのは手間がかかります．というのも，次の要素への参照しか持たないため，削除のときは「末尾の一つ手前」を先頭から求める必要があるからです．次回に紹介する双方向連結リストを使うと，簡単に末尾要素を削除できるようになります．

みやさか・でんと　miyadent@anet.ne.jp

リスト8　SlistElmt.java

```
public class SlistElmt {

    private Object mData;    // 要素が指す記録したいデータへの参照
    private SlistElmt mNext; // 隣の要素への参照

    private SlistElmt() { /* (empty) */ }

    // コンストラクタ iDataは要素が指すオブジェクト
    public SlistElmt(Object iData) {
        mData = iData;
        mNext = null;
    }

    // 要素が指すオブジェクトを返す
    public Object getData() {
        return mData;
    }

    // 次の要素への参照を得る
    public SlistElmt getNext() {
        return mNext;
    }

    // 次の要素への参照をiNextに変更する
    public void setNext(SlistElmt iNext) {
        mNext = iNext;
    }
}
```

リスト9　Slist.java（フィールド）

```
public class Slist {

    private int       mSize; // 登録している要素の数
    private SlistElmt mHead; // 先頭の要素への参照
    private SlistElmt mTail; // 末尾の要素への参照
```

リスト10　Slist.java（コンストラクタ）

```
    public Slist() {
        mSize = 0;
        mHead = mTail = null;
    }
```

リスト11　Slist.java（insertNextメソッド）

```
    // iElementの次にiDataを追加する
    // iElementがnullならリストの先頭をiDataにする
    public void insertNext(SlistElmt iElement,Object iData) {
        //追加したい要素を作成する
        SlistElmt aNewElement = new SlistElmt(iData);

        if(iElement == null){ //追加したい要素を先頭にしたい場合
            if(mSize == 0){   //まだ何も登録されていないなら
                //追加要素は末尾要素でもある
                mTail = aNewElement;
            }
            //追加要素の次を現時点の先頭要素とする
            aNewElement.setNext(mHead);
            mHead = aNewElement; //先頭要素は追加要素とする

        }else{ //追加したい要素を先頭以外にしたい場合
            SlistElmt aENext = iElement.getNext();
            if(aENext == null){ //iElementの次の要素がないなら
                //追加要素は末尾要素でもある
                mTail = aNewElement;
            }
            //追加要素の次をiElementの次とする
            aNewElement.setNext(aENext);
            //iElementの次を追加要素とする
            iElement.setNext(aNewElement);
        }
        //登録数を増やす
        ++mSize;
    }
```

リスト 12　Slist.java（removeNext メソッド）

```
    // iElement の次の要素を削除する
    // iElement が null なら先頭の要素を削除する
    // 戻り値は削除要素が指していたデータ
    public Object removeNext(SlistElmt iElement) {
        // 登録数が 0 なら削除できない
        if(mSize == 0){
            return null;
        }

        SlistElmt aOldElement; //削除される予定の要素

        if(iElement == null){  //先頭を削除する場合
            aOldElement = mHead;
            //先頭要素を現在の先頭の次にする
            mHead = mHead.getNext();
            if(mHead == null){ //先頭要素がなくなるなら
                mTail = null;  //末尾要素もなくなる
            }
        }else{ //先頭以外を削除する場合
            aOldElement = iElement.getNext();
            if(aOldElement == null){
                //次の要素が見つからないなら
                return null; //戻る
            }
            //iElement の新しい次の要素を得る
            SlistElmt aElmt = aOldElement.getNext();
            iElement.setNext(aElmt);
            if(aElmt == null){    //新しい次の要素がないなら
                mTail = iElement; //iElement は末尾要素である
            }
        }
        //登録数を減らす
        --mSize;
        //削除される要素のデータを返す
        return aOldElement.getData();
    }
```

リスト 13　Slist.java（そのほかのメソッド）

```
    // 登録数を返す
    public int getSize() {
        return mSize;
    }

    // 先頭要素を返す
    public SlistElmt getHead() {
        return mHead;
    }

    // 末尾要素を返す
    public SlistElmt getTail() {
        return mTail;
    }

    // iElement が先頭要素なら true を返す,
    // そうでないなら false を返す
    public boolean isHead(SlistElmt iElement) {
        return iElement == mHead;
    }

    // iElement が末尾要素なら true を返す,
    // そうでないなら false を返す
    public boolean isTail(SlistElmt iElement) {
        return iElement == mTail;
    }
}
```

```
Terminal — bash — 80x30

[07:14 PM  ~/Documents/Article-IF/No.10/sample-slist]
$ make
javac demo.java

[07:14 PM  ~/Documents/Article-IF/No.10/sample-slist]
$ java demo
* start *
# dump Slist : 1 2 3 4 5 6 7 : size = 7
# dump Slist : 2 3 4 5 6 7 : size = 6
# dump Slist : 2 3 4 5 6 : size = 5
* end *

[07:14 PM  ~/Documents/Article-IF/No.10/sample-slist]
$ 
```

図 4　Slist の利用例

リスト 14　Slist.java（Slist の利用例）

```
    //1～7 の Integer オブジェクトを登録した連結リストを返す
    private Slist makeSlistSample(){
        Slist aSlist = new Slist();
        for(int aI = 1; aI <= 7; aI++){
            Integer aIObj = new Integer(aI);
            aSlist.insertNext(aSlist.getTail(),aIObj);
        }
        return aSlist;
    }

    //連結リストの内容と登録数を表示する
    private void dumpSlist(Slist iSlist){
        System.out.print("# dump Slist :");
        SlistElmt aElmt = iSlist.getHead();
        while(aElmt != null){
            Object aObj = aElmt.getData();
            System.out.print(" ");
            System.out.print(aObj);
            aElmt = aElmt.getNext();
        }
        System.out.print(" : size = ");
        System.out.println(iSlist.getSize());
    }

    public void demo2(){
        //1～7 の Integer オブジェクトを登録する
        Slist aSlist = makeSlistSample();
        dumpSlist(aSlist);

        //先頭のオブジェクトを削除する
        aSlist.removeNext(null);
        dumpSlist(aSlist);

        //末尾の一つ手前を探す
        SlistElmt aElmt = aSlist.getHead();
        SlistElmt aTail = aSlist.getTail();
        SlistElmt aNext;
        while((aNext = aElmt.getNext()) != aTail){
            aElmt = aNext;
        }

        //末尾のオブジェクトを削除する
        aSlist.removeNext(aElmt);
        dumpSlist(aSlist);
    }
```

x86CPUだけでもマスタしたい

開発技術者のためのアセンブラ入門

第㉔回　SIMD命令（2）SSE/SSE2命令（その2）　大貫 広幸

今回は前回の続きとして，SSE/SSE2命令の各命令の動作について説明します．

SSE/SSE2命令の各命令の動作

SSE/SSE2で追加された命令を大きく分けると，表1に示した浮動小数点命令と，表2に示した浮動小数点以外の命令に分けられます．

SSE/SSE2の浮動小数点命令で使われるニモニックは，一定の規則に従いニモニック名が付けられています．そのため，その規則がわかれば，容易にニモニックから命令の動作を推測することができます．SSE/SSE2の浮動小数点命令のニモニックは，まず表3に示すような命令の種類を表す文字で始まります．次に，表4のような，その命令の動作を示す文字が付きます．そして最後に，表5のその命令が扱うデータの種類を表す文字が付きます．この規則を頭に入れて表1を見ると，各SSE/SSE2命令の動作の概要がニモニックから推測できると思います．

各SSE/SSE2命令を細かく分類すると，浮動小数点命令は「データ転送命令」,「変換命令」,「パックド単精度浮動小数点命令」,「シャッフル命令とアンパック命令」,「パックド算術命令」,

表1　SSE/SSE2の命令一覧（浮動小数点命令）

分類	データの構成	インストラクション名（ニモニック）	
		SSE命令	SSE2命令
データ転送命令	パックド	MOVAPS, MOVUPS, MOVHPS, MOVLPS, MOVHLPS, MOVLHPS, MOVMSKPS	MOVAPD, MOVUPD, MOVHPD, MOVLPD, MOVMSKPD
	スカラ	MOVSS	MOVSD
変換命令	パックド	CVTPI2PS, CVTPS2PI, CVTTPS2PI	CVTPI2PD, CVTPD2PI, CVTTPD2PI, CVTDQ2PD, CVTPD2DQ, CVTTPD2DQ, CVTPS2PD, CVTPD2PS
	スカラ	CVTSI2SS, CVTSS2SI, CVTTSS2SI	CVTSI2SD, CVTSD2SI, CVTTSD2SI, CVTSS2SD, CVTSD2SS
パックド単精度浮動小数点命令	パックド		CVTDQ2PS, CVTPS2DQ, CVTTPS2DQ
シャッフル命令とアンパック命令	パックド	SHUFPS, UNPCKHPS, UNPCKLPS	SHUFPD, UNPCKHPD, UNPCKLPD
パックド算術命令	パックド	ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, RSQRTPS, MAXPS, MINPS	ADDPD, SUBPD, MULPD, DIVPD, SQRTPD, MAXPD, MINPD
	スカラ	ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, RSQRTSS, MAXSS, MINSS	ADDSD, SUBSD, MULSD, DIVSD, SQRTSD, MAXSD, MINSD
比較命令	パックド	CMPPS	CMPPD
	スカラ	CMPSS, COMISS, UCOMISS	CMPSD, COMISD, UCOMISD
論理演算命令	パックド	ANDPS, ANDNPS, ORPS, XORPS	ANDPD, ANDNPD, ORPD, XORPD

表2　SSE/SSE2の命令一覧（浮動小数点以外の命令）

分　類	インストラクション名（ニモニック）	
	SSE命令	SSE2命令
64ビットSIMD整数命令	PAVGB, PAVGW, PEXTRW, PINSRW, PMOVMSKB, PSHUFW, PSADBW, PMULHUW, PMAXUB, PMAXSW, PMINUB, PMINSW	
128ビットSIMD整数命令		128ビット化されたMMX命令, MOVDQA, MOVDQU, MOVQ2DQ, MOVDQ2Q, PSHUFLW, PSHUFHW, PSHUFD, PUNPCKHQDQ, PUNPCKLQDQ, PADDQ, PSUBQ, PMULUDQ, PSLLDQ, PSRLDQ
キャッシュのフラッシュ		CLFLUSH
キャッシュ制御命令	MASKMOVQ, MOVNTQ, MOVNTPS	MASKMOVDQU, MOVNTDQ, MOVNTPD, MOVNTI
プリフェッチ命令	PREFETCHh	
命令順序付け命令	SFENCE	LFENCE, MFENCE
PAUSE		PAUSE
ステート管理	LDMXCSR, STMXCSR,	

「比較命令」,「論理演算命令」に分けることができます．また，浮動小数点以外のSSE/SSE2命令は「64ビットSIMD整数命令」,「128ビット整数SIMD命令」そして「制御に関する命令」に分けられます．

表3　SSE/SSE2の浮動小数点命令のニモニックで使われる命令の種類を示す文字

文字	命令の種類
MOV…	データ転送命令(DEST←SOU)
CVT…	変換命令，パックド単精度浮動小数点命令(DEST←変換(SOU))
SHUF…	シャッフル命令(DEST←シャッフル(SOU，DEST))
UNPCK…	アンパック命令(DEST←アンパック(SOU，DEST))
ADD…	パックド演算命令の加算(DEST←DEST + SOU)
SUB…	パックド演算命令の減算(DEST←DEST − SOU)
MUL…	パックド演算命令の乗算(DEST←DEST × SOU)
DIV…	パックド演算命令の除算(DEST←DEST ÷ SOU)
RCP…	パックド演算命令の逆数の近似値(DEST←近似値(1.0÷SOU))
SQRT…	パックド演算命令の平方根(DEST←SQRT(SOU))
RSQRT…	パックド演算命令の平方根の逆数の近似値(DEST←近似値(1.0÷SQRT(SOU)))
MAX…	パックド演算命令の最大値取得(DEST←MAX(DEST，SOU))
MIN…	パックド演算命令の最小値取得(DEST←MIN(DEST，SOU))
CMP…	DESTとSOUの比較結果(真偽)を，DESTに真(すべて1)偽(すべて0)の状態で返す比較命令
COMI…	SOU1とSOU2の比較結果をEFLAGS(ZF，PF，CF)に設定する比較命令．COMIはSOU1あるいはSOU2のSNaN,QNaNを無効操作例外とする
UCOMI…	SOU1とSOU2の比較結果をEFLAGS(ZF，PF，CF)に設定する比較命令．UCOMIはSNaNのみを無効操作例外とする
AND…	論理演算命令のビット単位の論理積(DEST←DEST and SOU)
ANDN…	論理演算命令のビット単位の否定論理積(DEST←(not DEST) and SOU)
OR…	論理演算命令のビット単位の論理和(DEST←DEST or SOU)
XOR…	論理演算命令のビット単位の排他的論理和(DEST←DEST xor SOU)

注：表中のDESTはdestination(先)，SOU，SOU1，SOU2はsource(元)

表5　SSE/SSE2の浮動小数点命令のニモニックで使われるデータを示す文字

文字	データ
PS	パックド単精度浮動小数点(四つの連続した単精度値)
PD	パックド倍精度浮動小数点(二つの連続した倍精度値)
SS	スカラ単精度浮動小数点(四つの連続した単精度値の最下位の値)単精度浮動小数点(単体で存在する単精度値)
SD	スカラ倍精度浮動小数点(二つの連続した倍精度値の最下位の値)倍精度浮動小数点(単体で存在する倍精度値)
PI	MMXレジスタ上あるいはメモリ上のパックド符号付きダブル・ワード整数(二つの連続した符号付きダブル・ワード整数値)
SI	32ビット汎用レジスタ上あるいはメモリ上の符号付きダブル・ワード整数(単体で存在する符号付きダブル・ワード整数値)
DQ	XMMレジスタ上あるいはメモリ上のパックド符号付きダブル・ワード整数(四つ，あるいは二つの連続した符号付きダブル・ワード整数値)

● データ転送命令

データ転送命令は，

- XMMレジスタ←XMMレジスタ
- XMMレジスタ←メモリ
- メモリ←XMMレジスタ

の転送をパックドおよびスカラで行うものです．

パックドの転送では，MOVAPS，MOVUPS，MOVAPD，MOVUPDの128ビットの転送命令のほかに，二つの単精度値あるいは一つの倍精度値，つまり64ビット長の転送命令も用意されています．MOVHPS，MOVLPS，MOVHLPS，MOVLHPSの4命令は，二つの単精度値を転送する命令で，MOVHPD，MOVLPDの2命令は，一つの倍精度値を転送する命令です．

特殊なデータ転送命令としてMOVMSKPS，MOVMSKPDがあります．この命令は，XMMレジスタ上の浮動小数点値の符号ビットのみを抽出し，CPUの汎用レジスタに転送するというものです．

スカラのデータ転送命令としては，MOVSS，MOVSDの命令を使用します．

このデータ転送命令の動作を図1に示します．

● 変換命令

変換命令は，

- 浮動小数点値←整数値
- 整数値←浮動小数点値
- 倍精度浮動小数点値←単精度浮動小数点値

表4　SSE/SSE2の浮動小数点命令のニモニックで使われる動作を示す文字

文字	動　作
A	128ビットのデータ転送で，メモリ上の128ビット値のアライメントが16バイトの倍数に整列している必要がある
U	128ビットのデータ転送で，メモリ上の128ビット値のアライメントが16バイトの倍数に整列している必要がない
H	64ビットのデータ転送で，XMMレジスタの上位64ビットのフィールドがアクセス対象
L	64ビットのデータ転送で，XMMレジスタの下位64ビットのフィールドがアクセス対象
HL	64ビットのデータ転送で，送り側をXMMレジスタの上位64ビットのフィールド，受け側をXMMレジスタの下位64ビットのフィールドとすることを意味する
LH	64ビットのデータ転送で，送り側をXMMレジスタの下位64ビットのフィールド，受け側をXMMレジスタの上位64ビットのフィールドとすることを意味する
MSK	XMMレジスタ上の浮動小数点の符号ビットのみをアクセス対象とする
2	変換で使用され，“to”の意味で使用．動作は，“2”の左辺の値を右辺のデータに変換することを示す．たとえば，「PS2PI」ならパックド単精度浮動小数点をパックド符号付きダブル・ワード整数に変換することになる
T	通常，高い精度の値を低い精度の値に変換する場合，レジスタMXCSRの丸め制御(RC)のビットの指定に従い，丸めを行う．しかし，浮動小数点を整数に変換するとき，ニモニックに‘T’が指定されている命令を使用すると，MXCSRのRCの指定に関係なく，切り捨てで整数に変換する

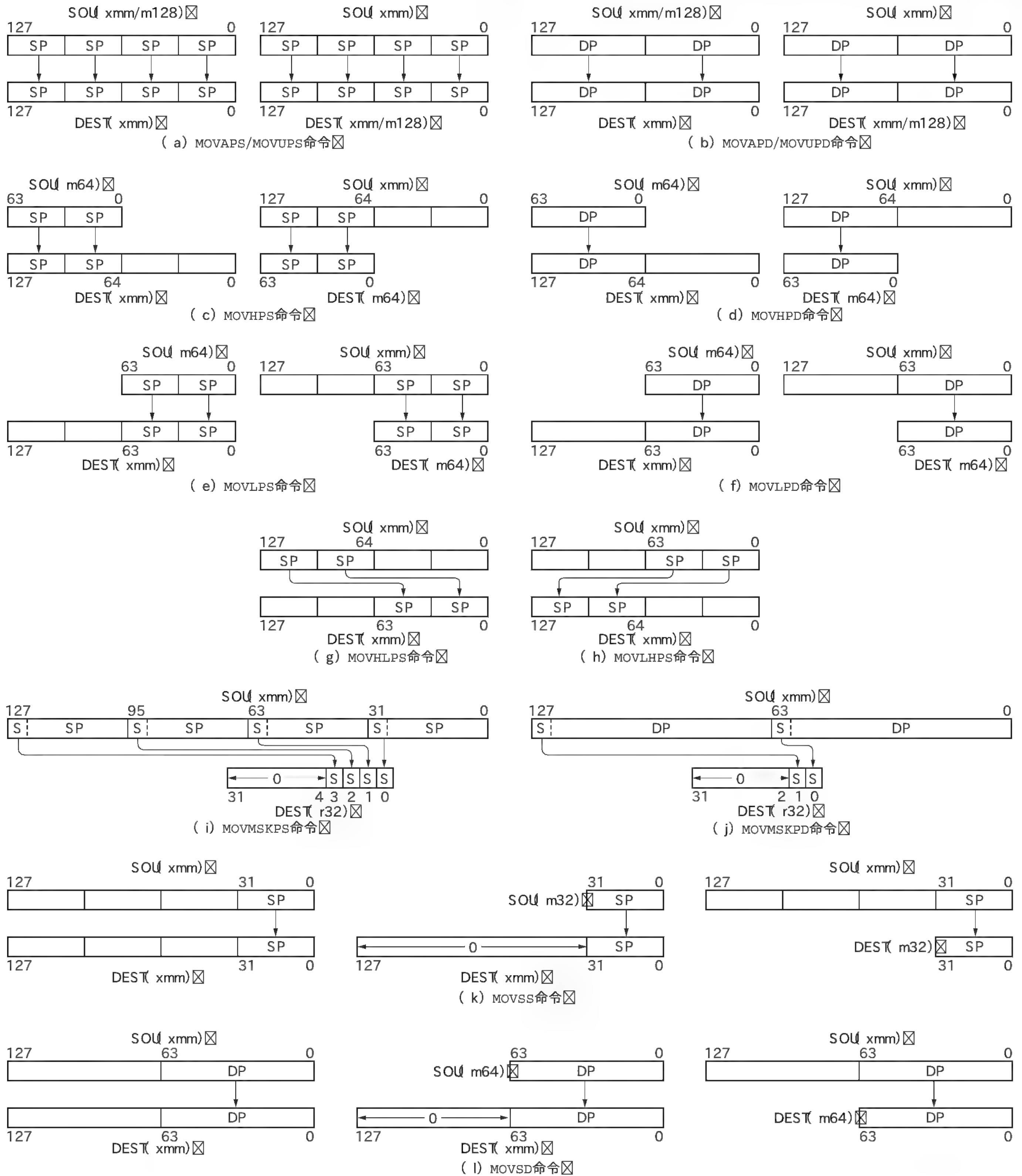


図1　データ転送命令の動作

●単精度浮動小数点値←倍精度浮動小数点値

の変換をパックドおよびスカラで行います．

「整数値←→浮動小数点」の変換では，整数値は符号付きダブル・ワード整数のみですが，浮動小数点値はSSEが単精度浮動小数点，SSE2が倍精度浮動小数点となっています．

パックドの変換では，CVTPI2PS，CVTPI2PD，CVTDQ2PDが「浮動小数点値←整数値」の変換，CVTPS2PI，CVTPD2PI，CVTPD2DQ，CVTTPS2PI，CVTTPD2PI，CVTTPD2DQが「整数値←浮動小数点値」の変換です．そして，CVTPS2PDが「倍精度浮動小数点値←単精度浮動小数点値」変換，CVTPD2PSが「単精度浮動小数点値←倍精度浮動小数点値」変換です．

スカラの変換では，CVTSI2SS，CVTSI2SDが「浮動小数点値←整数値」の変換，CVTSS2SI，CVTSD2SI，CVTTSS2SI，CVTTSD2SIが「整数値←浮動小数点値」の変換です．そして，CVTSS2SDが「倍精度浮動小数点値←単精度浮動小数点値」変換，CVTSD2SSが「単精度浮動小数点値←倍精度浮動小数点

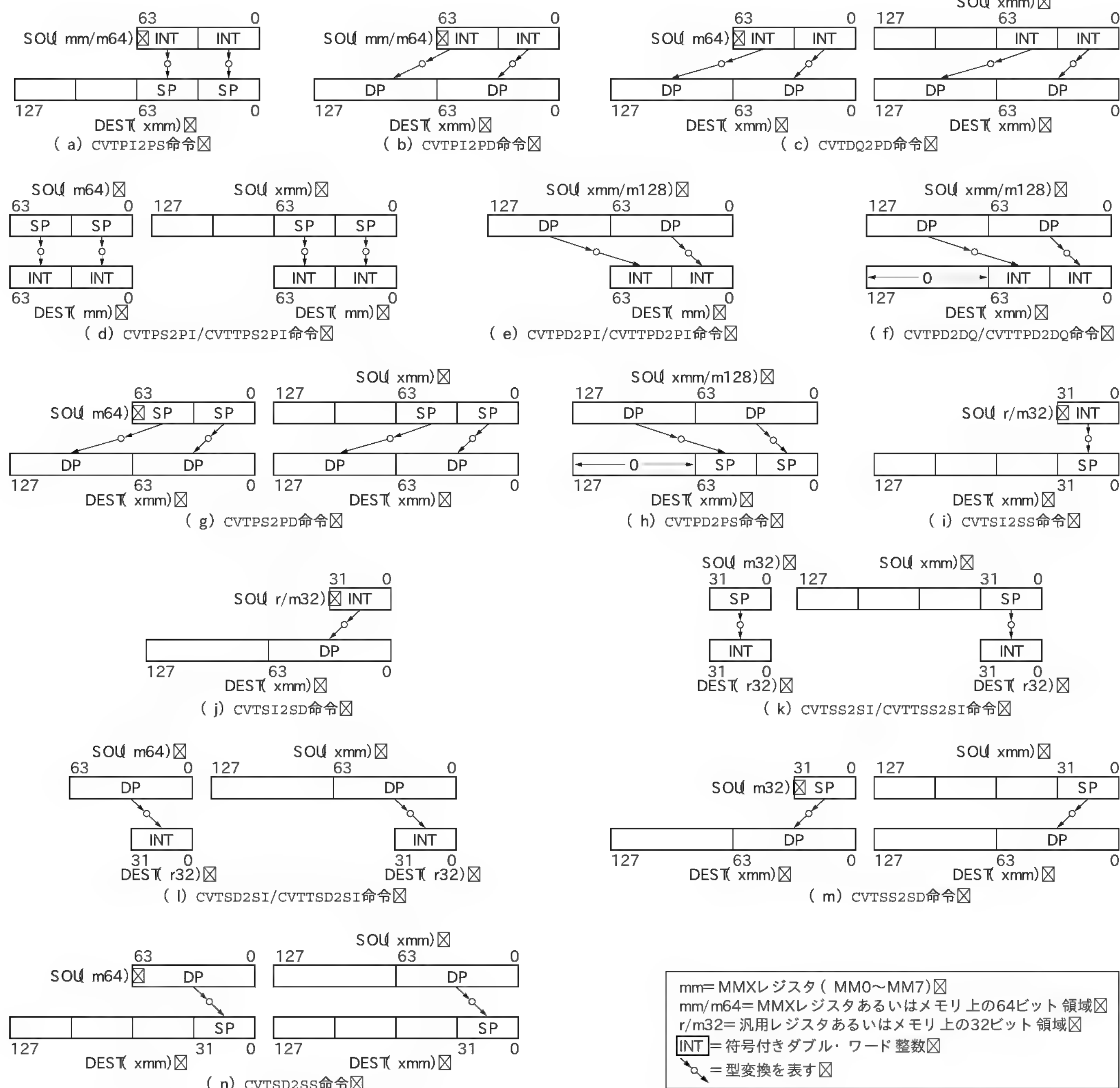


図2　変換命令の動作

値」変換です．

「整数値←浮動小数点値」の変換では，ニモニックのCVTの後にTが付いていない命令と付いている命令の2種類があります．Tが付いていない命令は，浮動小数点から整数に変換するとき，MXCSRレジスタの丸め制御（RC）の指定に従って丸めを行います．しかし，Tが付いている命令は，浮動小数点から整数に変換するとき，必ず切り捨てで丸めが行われます．

図2は，これら変換命令の動作を図で表したものです．

● パックド単精度浮動小数点命令

「パックド符号付きダブルワード整数←→パックド単精度浮動小数点」の変換を行います．

CVTDQ2PSは，XMMレジスタあるいはメモリの128ビット領域にある連続した4個の符号付きダブル・ワード整数値を，4個の単精度浮動小数点値に変換しXMMレジスタに格納します．

CVTPS2DQ，CVTTPS2DQは，XMMレジスタあるいはメモリの128ビット領域にある連続した4個の単精度浮動小数点値を，4個の符号付きダブル・ワード整数値に変換しXMMレジスタに格納します．この命令もTなしとT付きの2種類の命令があります．

図3は，パックド単精度浮動小数点命令の動作を表したものです．

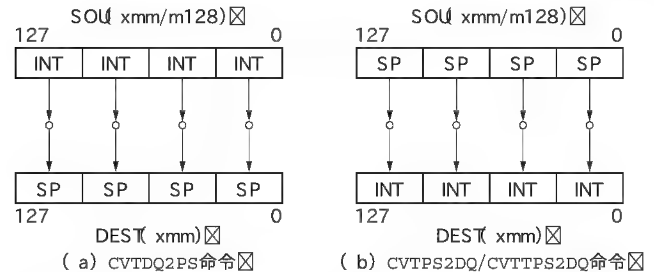


図3　パックド単精度浮動小数点命令の動作

● シャッフル命令とアンパック命令

SHUFPS，SHUFPDがシャッフル命令，UNPCKHPS，UNPCKLPS，UNPCKHPD，UNPCKLPDがアンパック命令です．

シャッフル命令は，パックされた浮動小数点の各フィールドの値を，指定したフィールドに移動するための命令です．シャッフル命令のオペランドは，インテル表記で，

DEST，SOU，imm8

の3オペランド形式になっています．imm8のイミディエイトで移動先のフィールドを指定します．

アンパック命令は，MMX命令のPUNPCKH…，PUNPCKL…を浮動小数点にしたものといえます．

シャッフル命令とアンパック命令の動作を図4に示します．

● パックド算術命令

パックド/スカラ，単精度/倍精度の浮動小数点は，四則（ADD…，SUB…，MUL…，DIV…）と平方根（SQRT…），最大値（MAX…），最小値（MIN…）の演算をもっています．

また，単精度浮動小数点のパックド/スカラのみですが，逆数（RCP…）と平方根の逆数（RSQRT…）の演算を行うことができます．

演算は，平方根と逆数，平方根の逆数の3命令が単項演算，そのほかが二項演算となります．

転送先をDEST，転送元をSOU，演算をopで表した場合，

```
DEST ← DEST op SOU
DEST ← op SOU
```

と演算され，DESTはXMMレジスタ，SOUはXMMレジスタあるいはメモリ上の値となります．

このパックド算術命令の動作を図にすると，図5（p.171）のようになります．

● 比較命令

比較命令は，浮動小数点（単精度あるいは倍精度）の比較結果を転送先に真偽の値で設定するCMP…の命令と，浮動小数点（単精度あるいは倍精度）の比較結果をCPUのEFLAGSレジスタに設定するCOMI…，UCOMI…の2種類があります．

図6（p.172）は，比較命令の動作を表したものです．

（1）**CMPPS，CMPPD，CMPSS，CMPSD**命令

CMP…命令は，パックド/スカラ両方で使用できます．CMP…命令は，転送先と転送元の各フィールドごと，指定方式で比較し，その結果が真なら転送先のそのフィールドのすべてのビットを1にし，偽なら転送先のそのフィールドをすべてのビットを0にします．

CMP…命令のオペランドは，インテル表記で，

DEST，SOU，imm8

の3オペランド形式になっています．imm8のイミディエイトで比較方法を指定します．

このimm8は「比較プレディケート」といい，「DEST:SOU」の比較方法を数字で指定するので，実用上不便な面もあります．そこでアセンブラによっては，ニモニックで比較方法を指定した2オペランドで記述する「疑似演算」と呼ぶ表記が使えるものもあります．

表6（p.171）は，CMP…命令の比較プレディケートとそれに対応する疑似演算を示したものです．

（2）**COMISS，COMISD，UCOMISS，UCOMISD**命令

COM…，UCOM…命令はスカラのみで，オペランドはインテル表記で，

SOU1，SOU2

となり，「SOU1－SOU2」の減算を行い，大小関係をEFLAGSレジスタのZF，PF，CFに設定します．比較の結果，アンオーダなら（ZF＝1，PF＝1，CF＝1）となります．また，「より大きい」なら（ZF＝0，PF＝0，CF＝0），「より小さい」なら（ZF＝0，PF＝0，CF＝1），「等しい」なら（ZF＝1，PF＝0，CF＝0）となります．

COM…とUCOM…命令の違いは，オペランドの値がNaN

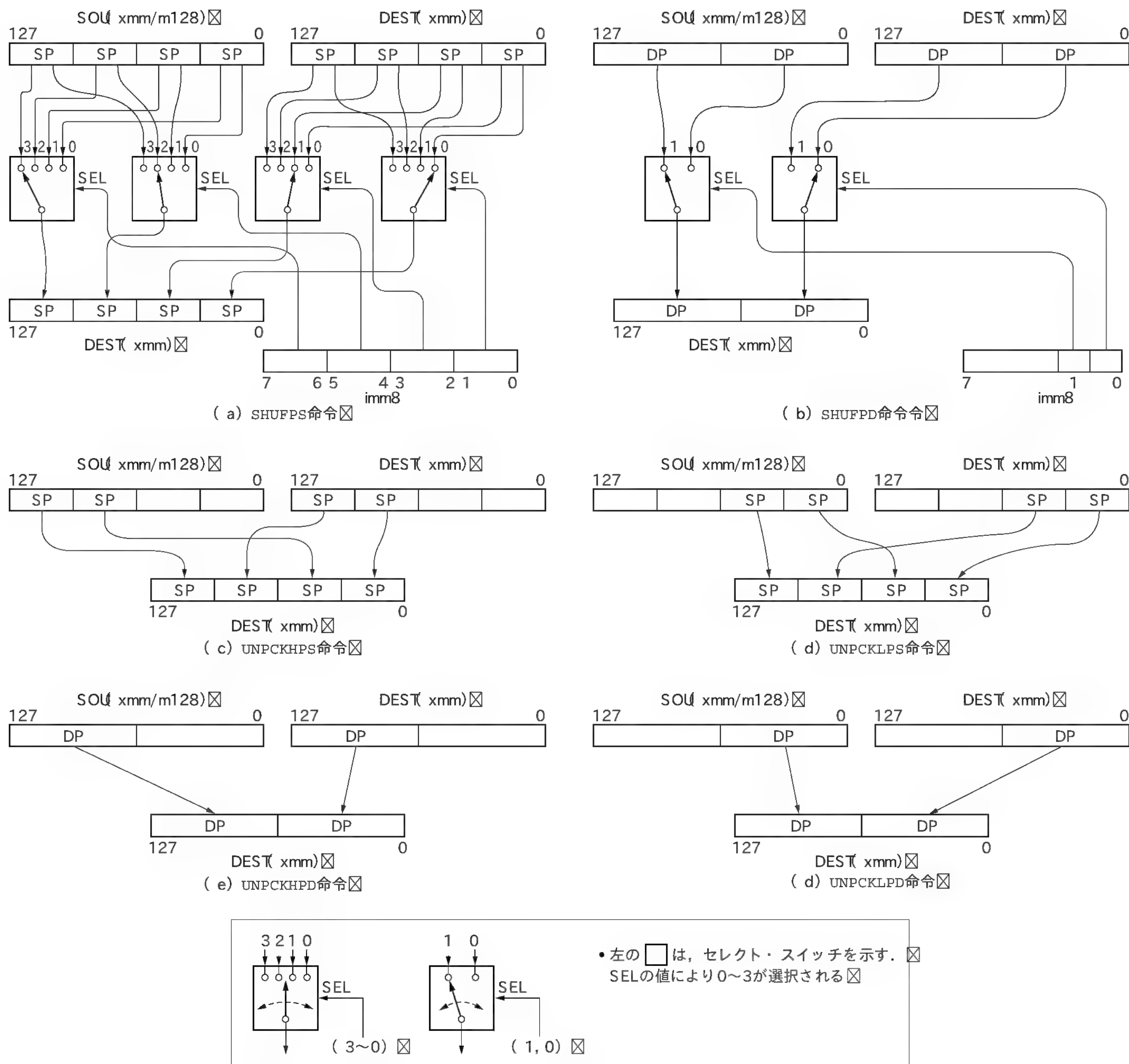


図4　シャッフル命令とアンパック命令

(SNaNとQNaN)のときの動作の違いです．COM…は，SNaNとQNaNの両方を無効操作例外とします．しかしUCOM…命令は，SNaNのみを無効操作例外とします．

● 論理演算命令

論理演算命令は，パックド浮動小数点(単精度，倍精度)のみ使用可能です．演算はビットごとに行われ，種類は論理積(AND)，否定論理積(ANDN)，論理和(OR)，排他的論理和(XOR)の4種類です．

MMX命令同様，SSE/SSE2命令にも否定(NOT)の演算はありません．しかし，MMX命令で追加された「DEST ←(not DEST) and SOU」の否定論理積がSSE/SSE2命令でも使用できます．

図7(p.172)は，論理演算命令の動作を表したものです．

● 64/128ビットSIMD整数命令

SSEでは，MMXレジスタを使用する64ビットSIMD整数命令が12個，MMX命令に追加されました．

さらにSSE2では，128ビットSIMD整数命とし て，XMMレジスタを使用する128ビット化されたMMX命令も使用可能となりました．また，それにともない14命令の追加がなされています．

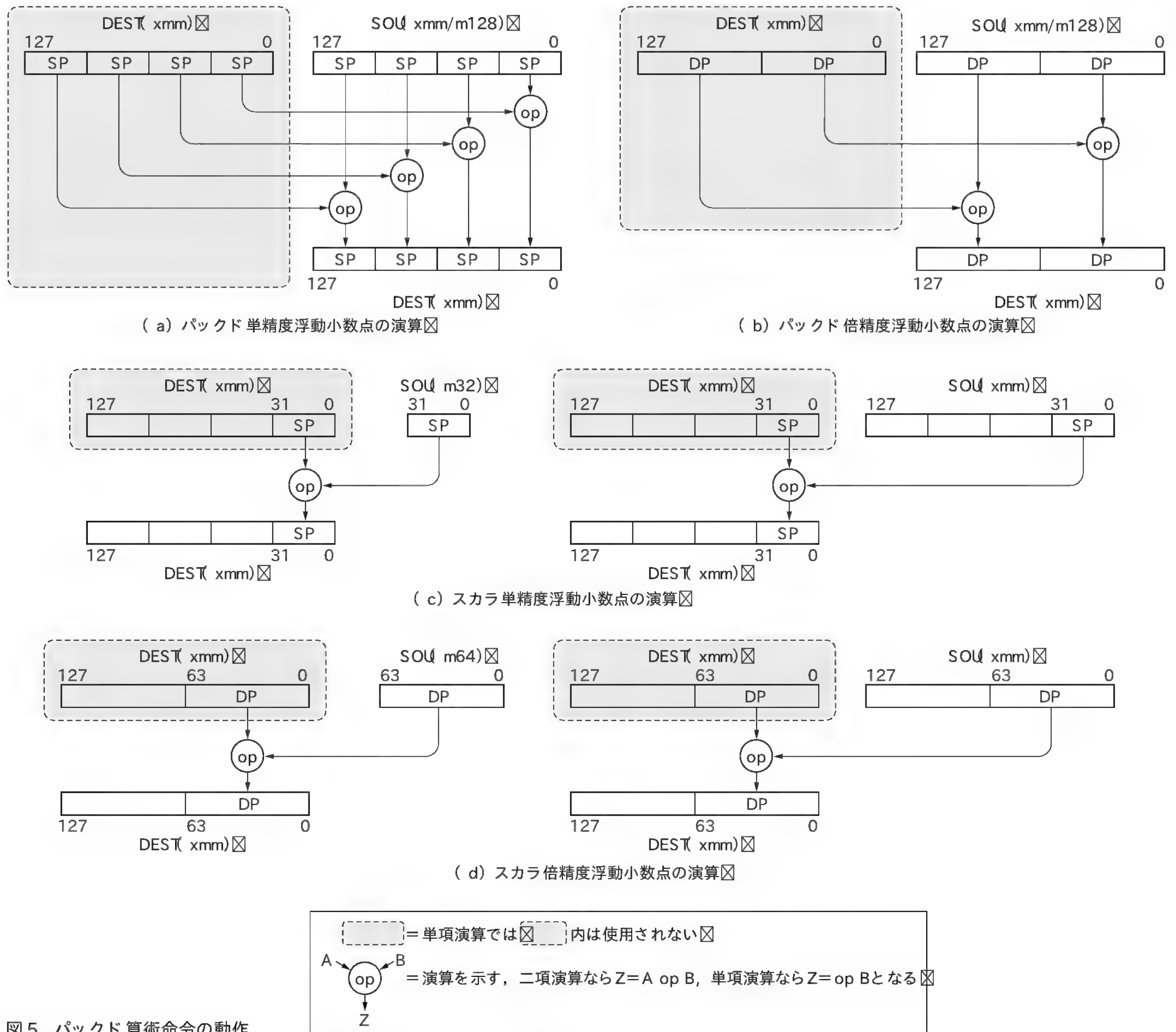
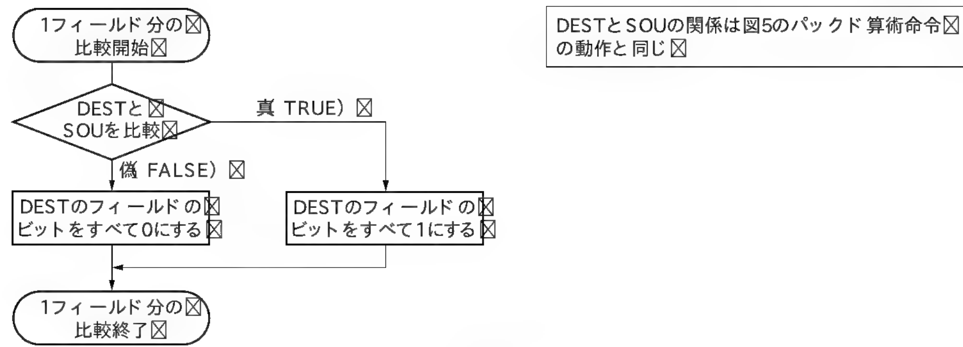


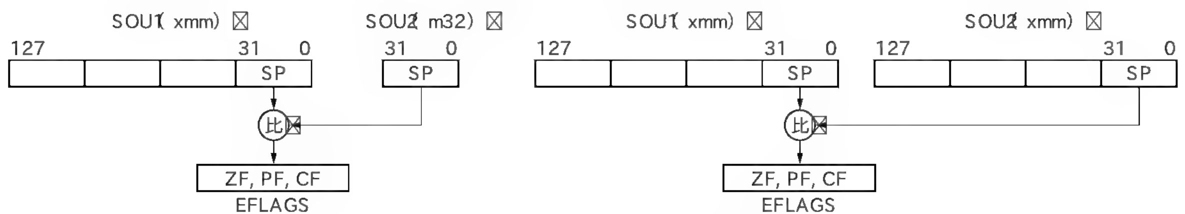
図5　パックド算術命令の動作

表6　SSE/SSE2のCMP…命令の比較プレディケートと疑似演算

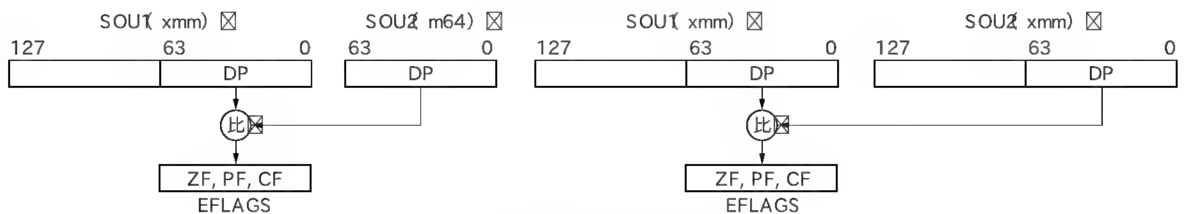
比較プレディケート	CMP…命令での表記	疑似演算での表記
0 (DEST＝SOUなら真) (DEST≠SOUなら偽)	CMPPS DEST,SOU,0 CMPPD DEST,SOU,0 CMPSS DEST,SOU,0 CMPSD DEST,SOU0	CMPEQPS DEST,SOU CMPEQPD DEST,SOU CMPEQSS DEST,SOU CMPEQSD DEST,SOU
1 (DEST＜SOUなら真) (DEST≧SOUなら偽)	CMPPS DEST,SOU,1 CMPPD DEST,SOU,1 CMPSS DEST,SOU,1 CMPSD DEST,SOU,1	CMPLTPS DEST,SOU CMPLTPD DEST,SOU CMPLTSS DEST,SOU CMPLTSD DEST,SOU
2 (DEST≦SOUなら真) (DEST＞SOUなら偽)	CMPPS DEST,SOU,2 CMPPD DEST,SOU,2 CMPSS DEST,SOU,2 CMPSD DEST,SOU,2	CMPLEPS DEST,SOU CMPLEPD DEST,SOU CMPLESS DEST,SOU CMPLESD DEST,SOU
3 (アンオーダなら真) (オーダなら偽)	CMPPS DEST,SOU,3 CMPPD DEST,SOU,3 CMPSS DEST,SOU,3 CMPSD DEST,SOU,3	CMPUNORDPS DEST,SOU CMPUNORDPD DEST,SOU CMPUNORDSS DEST,SOU CMPUNORDSD DEST,SOU
4 (DEST≠SOUなら真) (DEST＝SOUなら偽)	CMPPS DEST,SOU,4 CMPPD DEST,SOU,4 CMPSS DEST,SOU,4 CMPSD DEST,SOU,4	CMPNEQPS DEST,SOU CMPNEQPD DEST,SOU CMPNEQSS DEST,SOU CMPNEQSD DEST,SOU
5 (DEST≧SOUなら真) (DEST＜SOUなら偽)	CMPPS DEST,SOU,5 CMPPD DEST,SOU,5 CMPSS DEST,SOU,5 CMPSD DEST,SOU,5	CMPNLTPS DEST,SOU CMPNLTPD DEST,SOU CMPNLTSS DEST,SOU CMPNLTSD DEST,SOU
6 (DEST＞SOUなら真) (DEST≦SOUなら偽)	CMPPS DEST,SOU,6 CMPPD DEST,SOU,6 CMPSS DEST,SOU,6 CMPSD DEST,SOU,6	CMPNLEPS DEST,SOU CMPNLEPD DEST,SOU CMPNLESS DEST,SOU CMPNLESD DEST,SOU
7 (オーダなら真) (アンオーダなら偽)	CMPPS DEST,SOU,7 CMPPD DEST,SOU,7 CMPSS DEST,SOU,7 CMPSD DEST,SOU,7	CMPORDPS DEST,SOU CMPORDPD DEST,SOU CMPORDSS DEST,SOU CMPORDSD DEST,SOU



（a）CMPxx命令



（b）COMISS/UCOMISS命令



（c）COMISD/UCOMISD命令

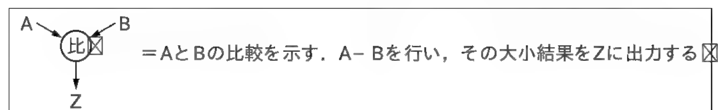


図6
比較命令の動作

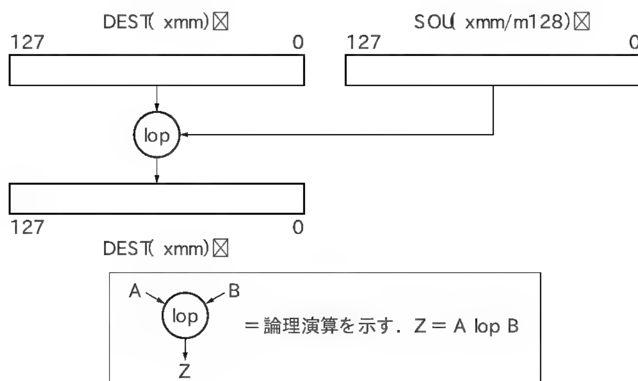


図7　論理演算の動作

表7（pp.174-175）は，既存のMMX命令，SSEで追加された64ビットSIMD整数命令，そしてSSE2で追加された128ビットSIMD整数命令を一覧にしたものです．また，一部動作が難解な命令について図8に示します．

● 制御に関する命令

制御に関する命令として，データのメモリ・アクセスを高速化するためのキャッシュ制御やプリフェッチの命令，命令の実行順序を決める命令などがあります．また，ステート管理命令としてMXCSRレジスタのロード/ストア，FPU関係のレジスタ（MMXを含む）やSSE関係のレジスタ全部のロード/ストアを行う命令があります．

（1）キャッシュ制御とプリフェッチ，命令順序付け

SSE/SSE2命令では，MMXやSSE/SSE2が64ビットや128ビットといった大きなデータを扱うためメモリのリード/ライトに時間がかかります．それをキャッシュやプリフェッチを制御することで，いくらかでも効率的に行おうとするのがこの命令です．

MASKMOV…命令は，MMXレジスタあるいはXMMレジスタ上のバイトをマスクで選択し，マスクで選択されたバイトのみをメモリにライトするという命令です．

MOVNT…命令は，すぐにリードされることのないデータのメモリ・ライトのとき，キャッシュにデータを残さずにメモリ・

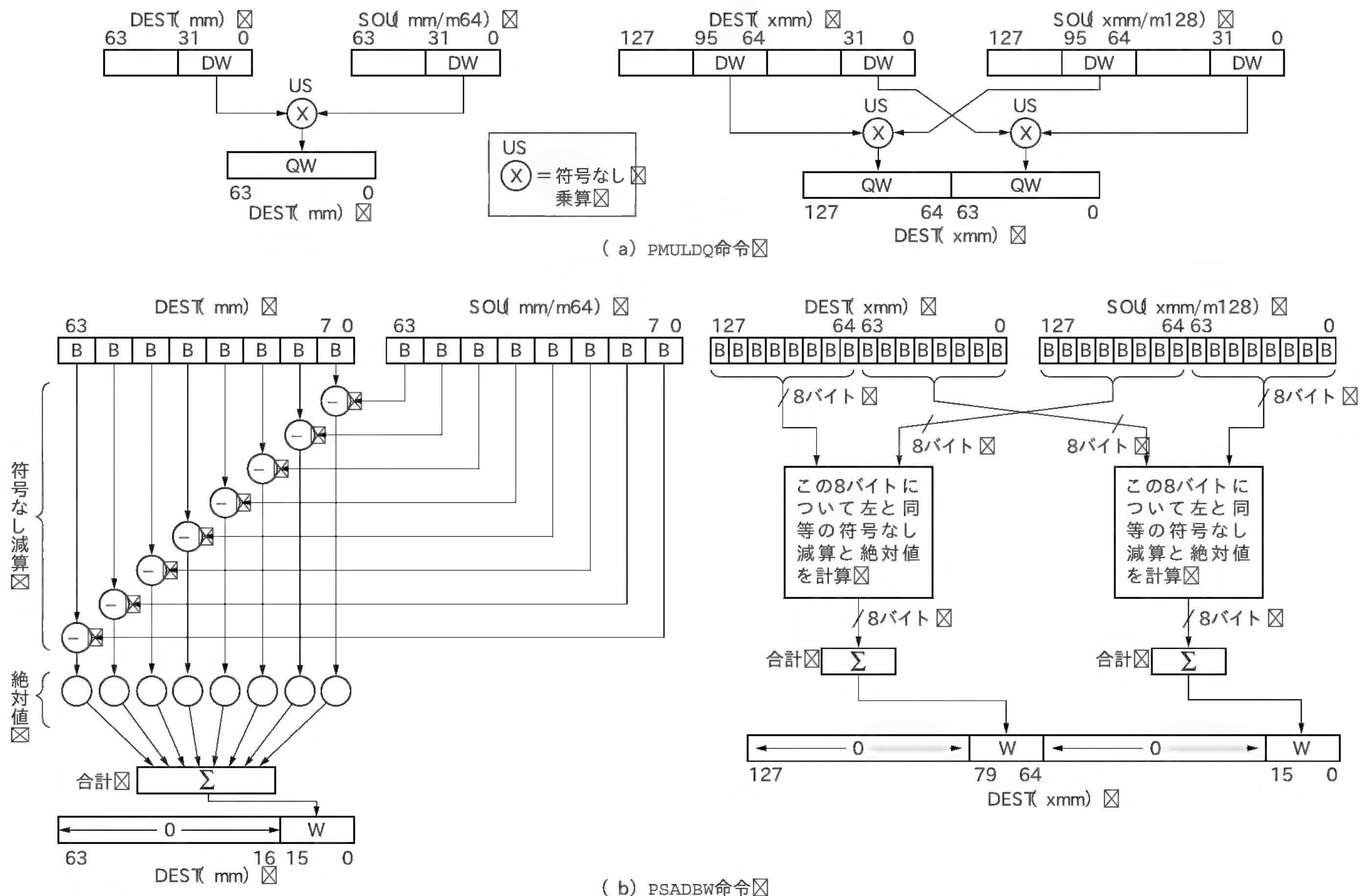


図8 SSE/SSE2命令で追加された動作が難解なSIMD整数命令

ライトするための命令です．これは，メモリ・リード時のキャッシュ・ヒットの効率を上げたいときに使用します．

PREFETCH命令は，実際にデータが必要になる前に，データをメモリからリードし，キャッシュに蓄えるための命令です．これにより，実際にデータが必要になったときのデータ・リードを速くすることができます．この命令を実行すると，CPUはメモリ・アクセスがない期間を利用して，PREFETCH命令で指定されたメモリからデータをリードします．

命令順序付け命令は，メモリのロード/ストアの順序を明確にする命令です．命令順序付け命令はフェンス命令とも呼ばれ，ストアのSFENCE，ロードのLFENCE，ロード/ストアのMFENCEの3命令があります．SFENCEはSSE，LFENCEとMFENCEがSSE2の命令です．フェンス命令を実行することで，フェンス命令後のストア，ロード，ロード/ストアの動作を，フェンス命令実行前のストア，ロード，ロード/ストアの完了を待ってから実行されることを保証します．

SSE2命令では，指定リニア・アドレスのキャッシュ・ラインの書き込みと無効化を行うCLFLASH命令，一定時間次の命令の実行を遅らせるPAUSE命令も追加されています．

(2) ステート管理命令

MXCSRレジスタのロードを行う命令としてLDMXCSR，ストアを行う命令としてSTMXCSRがあります．このLDMXCSR/STMXCSRの命令は，アプリケーションレベルのプログラムで使用可能です．

また，FPU関係のレジスタ(MMXを含む)とSSE関係のレジスタ全部のストア/ロードを行う命令としてFXSAVE命令とFXRSTOR命令があります．

このFXSAVE命令とFXRSTOR命令は，OSがタスクあるいはスレッドを切り替えるときに必要な，レジスタのセーブとリストアを行うためのものです．そのため，アプリケーション・レベルのプログラムでは，FXSAVE命令とFXRSTOR命令は使用しません．

*　　　*

次回はSSE/SSE2命令のMASM，gasでの使用方法について解説します．

おおぬき・ひろゆき　大貫ソフトウェア設計事務所

表7　MMX，SSE/SSE2で使用できる64/128ビット SIMD整数命令

インストラクション名（ニモニック）	MMXレジスタ（64ビット）	XMMレジスタ（128ビット）	備　考
	オペランド	オペランド	
MOVD	○ mm，r/m32 r/m32，mm	② xmm，r/m32 r/m32，xmm	
MOVQ	○ mm，mm/m64 mm/m64，mm	② xmm，xmm/m64 xmm/m64，xmm	クワッド・ワードの転送 転送先がXMMレジスタなら上位クワッド・ワードはゼロになる
MOVDQA	—	② xmm，xmm/m128 xmm/m128，xmm	ダブル・クワッド・ワードの転送
MOVDQU	—	② xmm，xmm/m128 xmm/m128，xmm	ダブル・クワッド・ワードの転送 メモリ上の128ビット値のアライメントが16バイトの倍数に合っている必要がない
MOVQ2DQ	② xmm，mm		MMXレジスタからXMMレジスタへのクワッド・ワードの転送 転送先のXMMレジスタの上位クワッド・ワードはゼロになる
MOVDQ2Q	② mm，xmm		XMMレジスタの下位クワッド・ワードをMMXレジスタへ転送
PACKSSWB PACKSSDW PACKUSWB	○ mm，mm/m64	② xmm，xmm/m128	
PUNPCKHBW PUNPCKHWD PUNPCKHDQ	○ mm，mm/m64	② xmm，xmm/m128	
PUNPCKHQDQ	—	② xmm，xmm/m128	転送先のxmmと転送元のxmm/m128の上位クワッド・ワードを転送先xmmにインタリーブする
PUNPCKLBW PUNPCKLWD PUNPCKLDQ	○ mm，mm/m64	xmm，xmm/m128	
PUNPCKLQDQ	—	② xmm，xmm/m128	転送先のxmmと転送元のxmm/m128の下位クワッド・ワードを転送先xmmにインタリーブする
PADDB PADDW PADDD	○ mm，mm/m64	② xmm，xmm/m128	
PADDQ	② mm，mm/m64	② xmm，xmm/m128	MMXレジスタではクワッド・ワード整数のアップ・アラウンド加算， XMMレジスタではパックド・クワッド・ワード整数のアップ・アラウンド加算
PADDSB PADDSW PADDUSB PADDUSW	○ mm，mm/m64	② xmm，xmm/m128	
PSUBB PSUBW PSUBD	○ mm，mm/m64	② xmm，xmm/m128	
PSUBQ	② mm，mm/m64	② xmm，xmm/m128	MMXレジスタではクワッド・ワード整数のアップ・アラウンド減算， XMMレジスタではパックド・クワッドワード整数のアップアラウンド減算
PSUBSB PSUBSW PSUBUSB PSUBUSW	○ mm，mm/m64	② xmm，xmm/m128	
PMULHUW	① mm，mm/m64	② xmm，xmm/m128	パックド符号なしワード整数の乗算を行い，積の上位16ビットをmmあるいはxmmに格納
PMULHW PMULLW PMADDWD	○ mm，mm/m64	② xmm，xmm/m128	
PMULUDQ	② mm，mm/m64	② xmm，xmm/m128	符号なしダブル・ワード整数のアップ・アラウンド乗算． 動作は図8参照
PAVGB PAVGW PSADBW	① mm，mm/m64	② xmm，xmm/m128	PAVG…は，四捨五入の丸めを用いたパックド符号なし整数の平均． PAVGBがバイト，PAVGWがワードの演算． PSADBWは，パックド符号なしバイト整数の差の絶対値の合計

表7　MMX，SSE/SSE2で使用できる64/128ビットSIMD整数命令（つづき）

インストラクション名（ニモニック）	MMXレジスタ（64ビット） オペランド	XMMレジスタ（128ビット） オペランド	備　考
PCMPEQB PCMPEQW PCMPEQD PCMPGTB PCMPGTW PCMPGTD	○ mm，mm/m64	② xmm，xmm/m128	
PAND PANDN POR PXOR	○ mm，mm/m64	② xmm，xmm/m128	
PSLLW PSLLD PSLLQ	○ mm，mm/m64 mm，imm8	② xmm，xmm/m128 xmm，imm8	
PSLLDQ	−	② xmm，imm8	バイト左シフト シフトするバイト数はimm8で指定
PSRLW PSRLD PSRLQ	○ mm，mm/m64 mm，imm8	② xmm，xmm/m128 xmm，imm8	
PSRLDQ	−	② xmm，imm8	バイト右シフト シフトするバイト数はimm8で指定
PSRAW PSRAD	○ mm，mm/m64 mm，imm8	② xmm，xmm/m128 xmm，imm8	
PEXTRW	① r32，mm，imm8	② r32，xmm，imm8	imm8で指定された位置のワードを，mmあるいはxmmから抽出しr32の下位ワードに転送．r32の上位ワードはゼロになる
PINSRW	① mm，r32/m16，imm8	② xmm，r32/m16，imm8	r32/m16の下位ワードを，mmあるいはxmmのimm8で指定された位置に挿入
PMAXUB PMAXSW PMINUB PMINSW	① mm，mm/m64	② xmm，xmm/m128	PMAX…は，パックド整数の最大値取得． PMIN…は，パックド整数の最小値取得． …UBは符号なしバイト整数，…SWが符号付きワード整数の演算
PMOVMSKB	① r32，mm	② r32，xmm	mmあるいはxmmの各バイトの最上位（符号）ビットを抽出し，バイト・マスクを作りr32に転送
PSHUFW	① mm，mm/m64，imm8	−	imm8の指定に基づき，mm/m64のワードをシャッフルし，mmに格納する
PSHUFHW PSHUFLW	−	② xmm，xmm/m128 imm8	imm8の指定に基づき，xmm/m128の4ワードをシャッフルし，xmmに格納する．PSHUFHWはこの処理を上位クワッド・ワードに対して行い，PSHUFLWはこの処理を下位クワッド・ワードに対して行う
PSHUFD	−	② xmm，xmm/m128，imm8	imm8の指定に基づき，xmm/m128のダブル・ワードをシャッフルし，xmmに格納する
EMMS	○	−	

注1：表中の−○①②は下記の意味をもつ．
　−：使用不可，○：SEE2，SSE，MMXで使用可能
　①：SSE2，SSEで使用可能，②：SSE2のみで使用可能
注2：オペランドはインテル表記で表し，下記の記号でアクセス対象を表す．
　imm8　：8ビットのイミディエイト値
　r32　：32ビット汎用レジスタ（EAX，EBX，ECX，EDX，…）
　mm　：MMXレジスタ（MMX0～MMX7）
　xmm　：XMMレジスタ（XMM0～XMM7）
　r/m32　：32ビット汎用レジスタあるいはメモリ上の32ビット領域
　r32/m16　：32ビット汎用レジスタの下位ワードあるいはメモリ上の16ビット領域
　mm/m64　：MMXレジスタあるいはメモリ上の64ビット領域
　xmm/m64：XMMレジスタの下位クワッド・ワードあるいはメモリ上の64ビット領域
　xmm/m128：XMMレジスタあるいはメモリ上の128ビット領域
　（MOVDQU命令を除きメモリ上の128ビット領域は，アライメントが16バイトの倍数に合っている必要がある．もしこれが合っていないと一般保護例外のエラーとなる）

TMS320C6713搭載DSPスタータ・キットを使った

C++による DSPオブジェクト指向プログラミング

第3回 アナログ信号入出力用クラスを使う簡単なプログラム(後編)

◆三上 直樹

今回は，アナログ信号の入出力を割り込みを使って行うプログラムと，ディジタル・フィルタのプログラムを示します．ディジタル・フィルタを実現する際には，オブジェクト指向プログラミングの中で重要な概念の一つであるポリモーフィズム(polymorphism)注1を利用して，FIRフィルタのプログラムを作成します．

1 割り込み方式によりアナログ信号入出力を行う簡単なプログラム

前回はポーリング方式を使いましたが，今回は割り込みを使って，A-D変換器からの入力信号をそのままD-A変換器に出力し，それと並行してボードのLEDを点滅させるプログラムを作成します．点滅のタイミングにはタイマ割り込みを使います．なお，このプログラムでは割り込みを使うので，新たに割り込みベクタを設定するためのアセンブリ言語で記述したファイルを作る必要があります．

● C++によるソース・ファイル

リスト1にソース・プログラム(`through_intr.cpp`)を示します．割り込みを使う場合は，`AIC23_Intr.hpp`をインクルードします．`dsk6713_led.h`はボード・サポート・ライブラリ(BSL)の中で，ボードのLEDをコントロールする関数のためのヘッダ・ファイルで，`csl_timer.h`はチップ・サポート・ライブラリ(CSL)[1]の中で，タイマをコントロールする関数のためのヘッダ・ファイルです．

グローバル領域で最初に宣言されている`IntrCfTbl[]`は，割り込みに関する設定を行うための構造体の配列です．この配列を`AIC23_Intr`クラスのコンストラクタに渡すことにより，McBSP1の受信割り込みはDSP割り込みの`INT11`に，タイマ0の割り込みはDSP割り込みの`INT14`に割り付けられます．ここで使われている`IRQ_EVT_RINT1`と`IRQ_EVT_TINT0`は，チップ・サポート・ライブラリの中で定義されている定数です．

次に，`AIC23_Intr`クラスのオブジェクトとして`codec`が宣言されています．メイン・プログラム内でこの宣言を行わない理由は，`AIC23_Intr`クラスのメンバ関数は`main`の外部にある関数`AIC_RX_ISR()`で呼ばれているからです．

▶ メイン・プログラム

最初にタイマ用ハンドルのための変数`hTimer`が宣言されています．

次の3行はタイマをコントロールするためのチップ・サポート・ライブラリのAPI関数[1]で，これでタイマの設定を行っています．

関数`TIMER_open()`はタイマ0をオープンします．

次の関数`TIMER_configArgs()`ではタイマのレジスタを設定します．第2引き数の`0x2C0`という設定により，タイマ・クロックのソースがDSPのクロックの1/4になるように設定されます．このボードの場合，DSPのクロックは225MHzなので，タイマのカウンタは17.78nsごと注2にカウント・アップされます．第3引き数の`56250`という設定により，カウンタがこの数に等しくなったときにタイマ割り込みを発生します．したがって，この設定により1msごと注3にタイマ割り込みを発生することになります．

関数`TIMER_start()`によりタイマのカウンタはカウントを開始します．

関数`DSK6713_rset()`は，ボードに搭載されているCPLDに設けられたレジスタの設定を行う関数で，第1引き数で指定されているレジスタに0を設定することにより，ボードに載っている4個のユーザLEDは消灯します．

以上の記述で，初期設定が終わります．次に，関数`IRQ_globalEnable()`でグローバル割り込みを許可すると，`AIC23_Intr`クラスのコンストラクタで設定されたMcBSP1の受信割り込みと，タイマ0割り込みが有効になります．

最後に`while`文の無限ループで割り込み待ちの状態に入ります．

▶ 割り込みサービス・ルーチンの書きかた

割り込みが発生したときに実行される関数を割り込みサービス・ルーチン(ISR：interrupt service routine)と呼びます．これをC/C++で記述する場合，必ず次の3点に従う必要があります．

注1：多態性または多相性とも呼ぶ．

注2：$4/(225\times10^{6})\cong17.78\times10^{-9}$(s)=17.78(ns)

注3：$56250\times4/(225\times10^{6})=1\times10^{-3}$(s)=1(ms)

(1)関数名の先頭に，必ずキーワード interrupt を付ける
(2)関数は引き数を持つことはできない
(3)関数の戻り値は必ず void でなければならない

▶ **AIC_RX_ISR()**

この関数で行っている処理は，TLV320AIC23のA-D変換器から送られてきた2チャネルのデータを読み込み，そのデータを何も処理をせず，そのままTLV320AIC23のD-A変換器へ送るというものです．

▶ **TIMER0_ISR()**

この関数は，タイマ0の割り込み発生時に実行される関数です．タイマ0の割り込みは1msごとに発生するように設定しています．そこで，この関数はタイマ割り込みの回数をカウントし，100回カウントしたところでLEDの表示を変更するという処理を行っています．つまり，LEDの表示は0.1秒ごとに更新されます．

変数 LEDs と count は static 変数として宣言していることに注意してください．これらを static 変数にしている理由は，ふたたび関数 TIMER0_ISR() が呼ばれたとき，この二つの変数が前回の値を保持している必要があるからです．

ボード上にある4個のLEDの表示は変数 LEDs の下位4ビットに対応し，1になったビットに対応するLEDが点灯します．

● リセット/割り込み用アセンブラ・ファイル

前回に掲載したプログラム(リスト2：前回はリスト1として掲載)にはリセットに対応するベクタの記述しかないので，INT11 と INT14 に対応する部分も追加する必要があります．これらを追加したものがリスト3です．

二つの割り込みに対して，「.ref」というアセンブラ指示命令によりエントリ・ポイントを宣言します．エントリ・ポイントを示すシンボルは，C++で書かれた割り込みサービス・ルーチンの関数名の先頭に「_」(アンダ・スコア)を付け，関数名の末尾に「__Fv」を付けます[注4]．したがって，以下のようになります．

関数名	エントリ・ポイントのシンボル
AIC_RX_ISR	_AIC_RX_ISR__Fv
TIMER0_ISR	_TIMER0_ISR__Fv

INT11，INT14 に対応する処理では，最初にレジスタ A0 の値をスタックにプッシュ(push)します[注5]．これは，割り込みサービス・ルーチンに分岐する際の分岐先のアドレスとして A0 を使用するからです．次に，MVKL と MVKH の二つの命令でレジスタ A0 に割り込みサービス・ルーチンのアドレスを設定し，

注4：Cの場合は本文で説明した命名規則とは異なっている．C++の場合でも，コンパイラのバージョンが変われば命名規則が変わる場合がある．これを調べる際には，コンパイラの生成したアセンブリ・ファイルを見ればよい．通常，アセンブリ・ファイルはビルドの後に消されるが，コンパイラ・オプションのAssemblyに関する箇所の設定により，アセンブリ・ファイルを消さずに残しておくことも可能．

注5：TMS320C6000ファミリ用のC/C++コンパイラでは，レジスタB15をスタック・ポインタとして使用する．

リスト1　割り込み方式によるA-D変換されたデータをそのままD-A変換するのと並列的にボードのLEDを点滅させるプログラム(through_intr.cpp)

```
// ------------------------------------------------
//    割り込みで AD 変換されたデータをそのまま DA 変換する
//    これと並列的に LED を点滅する
// ------------------------------------------------

#include "AIC23_Intr.hpp"
#include <dsk6713_led.h>        ← LED操作のためのボード・サポート・ライブラリ(BSL)用

#include <csl_timer.h>          ← タイマのためのチップ・サポート・ライブラリ(CSL)用

// 割り込み設定用のデータ
const AIC23_Intr::IntrConfig IntrCfTbl[] =
    {   {IRQ_EVT_RINT1, 11},    ← McBSP1受信割り込み用
        {IRQ_EVT_TINT0, 14},    ← タイマ0割り込み用
        {0, 0} };               ← ここで割り込み設定用データの終わり
AIC23_Intr codec(IntrCfTbl);

int main()
{
    TIMER_Handle hTimer;

    hTimer = TIMER_open(TIMER_DEV0, 0);
    TIMER_configArgs(hTimer, 0x02C0, 56250, 0);
    TIMER_start(hTimer);

    DSK6713_rset(DSK6713_USER_REG, 0);   ← 全部のLEDを消灯

    IRQ_globalEnable();                  ← グローバル割り込みの許可

    while(1) {}                          ← 割り込み待ち
}

// McBSP1 の受信割り込み用割り込みサービス・ルーチン
interrupt void AIC_RX_ISR()
{
    short ch[2];

    codec.Read(ch);
    codec.Write(ch);
}

// タイマ 0 の受信割り込み用割り込みサービス・ルーチン
// 割り込み周期:  1 ms
interrupt void TIMER0_ISR()
{
    static int LEDs = 0;
    static int count = 0;

    if (++count >= 99)
    {
        DSK6713_rset(DSK6713_USER_REG, LEDs);
        LEDs = ++LEDs & 0xF;
        count = 0;
    }
}
```

リスト2　アセンブリ言語によるリセット・ベクタに関する記述(vects_Reset.asm)

```
; ***********************************
;          Vector for Reset
; ***********************************
        .sect "vectors"
        .ref  _c_int00          ← C/C++ のエントリ・ポイントのシンボル

RESET:                          ← 以下リセット・ベクタ
        MVKL  _c_int00,A0  }
        MVKH  _c_int00,A0  }    ← A0に_c_int00に対応するアドレスをロード
        B     A0                ← A0で示されるアドレスへ分岐
        NOP
        NOP
        NOP
        NOP
        NOP
```

リスト 3　アセンブリ言語によるリセット・ベクタおよび割り込みベクタに関する記述（vects_Reset_INT11_14.asm）

```
; ***********************************************
;          Vector for Reset and Interrupt
; ***********************************************
        .sect "vectors"              ← C/C++のエントリ・ポイントのシンボル
        .ref  _c_int00
        .ref  _AIC_RX_ISR__Fv        ← McBSP1の受信割り込みに対する割り込みサービス・ルーチンのシンボル

        .ref  _TIMER0_ISR__Fv        ← タイマ0の割り込みに対する割り込みサービス・ルーチンのシンボル

RESET: ; リセット・ベクタ
        MVKL  _c_int00,A0            ← A0に_c_int00に対応するアドレスをロード
        MVKH  _c_int00,A0
        B     A0                     ← A0で示されるアドレスへ分岐
        NOP
        NOP
        NOP
        NOP
        NOP

        .space 0x20*10               ← 0x140バイトの領域を予約

; McBSP1の受信割り込みに対するベクタ ← 先頭アドレス= 0x00000160
INT11:
        STW   A0,*B15--[1]           ← A0をスタックへプッシュ
        MVKL  _AIC_RX_ISR__Fv,A0     ← A0に，_AIC_RX_ISR__Fvに対応するアドレスをロード
        MVKH  _AIC_RX_ISR__Fv,A0
        B     A0                     ← McBSP1の受信割り込みに対する割り込みサービス・ルーチンへの分岐

        LDW   *++B15[1],A0
        NOP
        NOP
        NOP   2                      ← スタックからA0をポップ

        .space 0x20*2                ← 0x40バイトの領域を予約

; タイマ0割り込みに対するベクタ     ← 先頭アドレス=0x000001C0
INT14:
        STW   A0,*B15--[1]           ← A0をスタックへプッシュ
        MVKL  _TIMER0_ISR__Fv,A0     ← A0にTIMER0_ISR__Fvに対応するアドレスをロード
        MVKH  _TIMER0_ISR__Fv,A0
        B     A0                     ← タイマ0割り込みに対する割り込みサービス・ルーチンへの分岐
        LDW   *++B15[1],A0
        NOP
        NOP
        NOP   2                      ← スタックからA0をポップ
```

（a）直接形FIRフィルタのブロック図

（b）転置形FIRフィルタのブロック図

※ z^{-1} は遅延器を表す

図1　FIRフィルタの代表的な構成方法

そのアドレスへB命令で分岐します．最後に，スタックに退避しておいた値をポップ（pop）してA0に戻します．

最後のNOP命令は，前回説明したようにB命令が遅延分岐命令になっているので，その遅延スロットを埋めるためのものです．ところで，C6000ファミリのDSPでは一つの割り込みベクタに対して8命令分の領域が割り当てられているため，この場合には，LDW命令を含めて4スロット分しか埋めることができません．そこで，いちばん最後のNOP命令のオペランドに2を付け加え，NOP 2と記述しています．これにより，NOP命令が2回実行されることになるので，これで五つの遅延スロットを埋めることができます．

2　FIRフィルタのプログラム

ここではクラスを利用してディジタル・フィルタのプログラムを作成します．作成するディジタル・フィルタはFIRフィルタとし，二つの異なった構造のFIRフィルタをオブジェクト指向プログラミングの中の重要な概念であるポリモーフィズムを使って実現します．

● FIRフィルタについて

最初に，FIRフィルタとその構成方法について簡単に説明します．FIRフィルタとはインパルス応答の継続時間が有限なフィルタです．FIRフィルタの入力信号を $x[n]$，出力信号を $y[n]$，フィルタの係数を h_m（$m=0, 1, \cdots, M$）とすると，次の式[注6]で表すことができます．

$$y[n]=\sum_{m=0}^{M} h_m x[n-m] \quad \cdots\cdots (1)$$

この式で，Mをこのフィルタの次数と言います．また，このフィルタの伝達関数 $H(z)$ は次のようになります．

$$H(z)=\sum_{m=0}^{M} h_m z^{-m} \quad \cdots\cdots (2)$$

このフィルタを実現する場合，いくつかの構成方法があります．代表的なのは直接形（direct form）と転置形（transposed form）です．それらのブロック図を図1（a），（b）に示します．

直接形のブロック図は，式（1）の差分方程式と1対1に対応しています．一方，転置形のブロック図は，式（1）を次のように変形したものに対応します．

$$\begin{cases} u_M[n]=h_M x[n] \\ u_{M-1}[n]=h_{M-1}x[n]+u_M[n-1] \\ \quad\vdots \\ u_1[n]=h_1 x[n]+u_2[n-1] \\ u_0[n]=h_0 x[n]+u_1[n-1] \\ y[n]=u_0[n] \end{cases} \quad \cdots\cdots (3)$$

注6：このような式を差分方程式（difference equation）と呼ぶ．

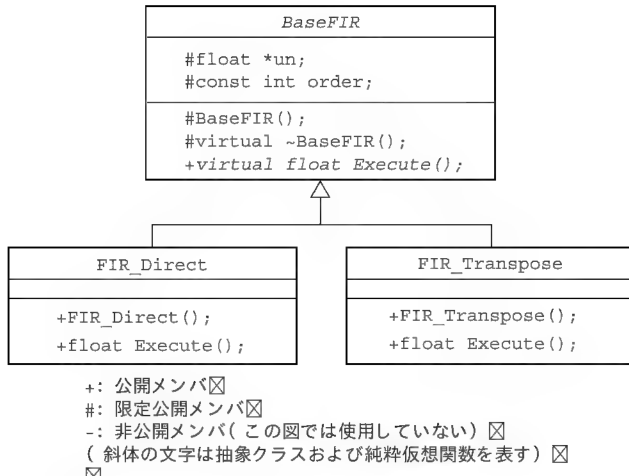


図2　FIRフィルタ実現のためのクラスの階層構造

● FIRフィルタのクラス

FIRフィルタの処理は，初期化の部分とフィルタ処理の本体の二つに大きく分けることができます．この初期化の部分は，直接形および転置形ともに同じ処理で実現できます．そこで，初期化の部分を基底クラスBaseFIRとし，これを継承するかたちで直接形と転置形それぞれのフィルタ処理の部分を派生クラスとして作成します．これをリスト4に示します．また，この継承のようすをUMLのクラス図風[注7]に表したものを図2に示します．

▶ 基底クラス

基底クラスBaseFIRは派生することを前提とするため，抽象クラスとします．ここではポリモーフィズムを使った例としてFIRフィルタを実現するので，フィルタを実行するためのメンバ関数Execute()は仮想関数にします．さらに，抽象クラスにするためには純粋仮想関数を一つ以上含む必要があるので，仮想関数Execute()を純粋仮想関数として宣言しています．

コンストラクタでは二つのことを行います．一つはnew演算子を使い，図1の遅延器の相当する領域を確保します[注8]．フィルタの次数は，非公開メンバであるorderにコンストラクタのメンバ初期設定[注9]によって値が設定されており，設定する領域の大きさはorder+1になります．次に，この領域に0.0を書き込み，初期化を行います．

デストラクタでは，new演算子で確保した領域をdelete演算子で解放します．派生クラスに対する基底クラスのポインタに，delete演算子を使って明示的にそのオブジェクトを放棄する場合，基底クラスのデストラクタが仮想関数でなければ，派生クラスのデストラクタは実行されません．したがって，派生クラスのデストラクタでもオブジェクトの破棄を行っている場合，その破棄が実行されないことになります．これを防ぐためには基底クラスのデストラクタは仮想関数にする必要があります(2)．

リスト4　FIRフィルタ用のクラス(FIR_class.cpp)

```
// -----------------------------------------------------
//      FIRフィルタ用のクラス
// -----------------------------------------------------

// ------------------------------------------------
//    FIRフィルタの基底クラス
// ------------------------------------------------
class BaseFIR
{
protected:
    float *un;
    const int order;
    BaseFIR(int nOrd);
    virtual ~BaseFIR() { delete[] un; }
public:
    virtual float Execute(const float hm[],
                          const float xin) = 0;
};

// BaseFIRのコンストラクタ
BaseFIR::BaseFIR(int nOrd) : order(nOrd)
{
    un = new float[order+1];
    for (int k=0; k<=order; k++) un[k] = 0.0;
}
// ------------- BaseFIR クラスの終了 --------------------

// ------------------------------------------------
//    BaseFIRの派生クラス，直接形
// ------------------------------------------------
class FIR_Direct : public BaseFIR
{
public:
    FIR_Direct(int nOrd) : BaseFIR(nOrd) {}
    float Execute(const float hm[], const float xin);
};

float FIR_Direct::Execute(const float hm[], const float xin)
{
    float acc = 0.0;
    un[0] = xin;
    for (int k=0; k<order; k++) acc = acc + hm[k]*un[k];
    for (int k=order; k>0; k--) un[k] = un[k-1];
    return acc;
}
// ------------- FIR_Direct  クラスの終了-------------

// ------------------------------------------------
//    BaseFIRの派生クラス，転置形
// ------------------------------------------------
class FIR_Transpose : public BaseFIR
{
public:
    FIR_Transpose(int nOrd) : BaseFIR(nOrd) {}
    float Execute(const float hm[], const float xin);
};

float FIR_Transpose::Execute(const float hm[], const float xin)
{
    for (int k=0; k<order; k++) un[k] = hm[k]*xin + un[k+1];
    un[order] = hm[order]*xin;
    return un[0];
}
// ------------- FIR_Transpose  クラスの終了----------
```

注7：UMLのクラス図では，属性(C++ではデータ・メンバのこと)を表す場合に，「名前：タイプ」という書きかたを行うが，ここではC++に準じた書きかたを採用した．ほかにも本来のクラス図の書きかたとは異なる部分があるので，「クラス図」とは書かずに「クラス図風」と書いている．

注8：このプログラムでは，リスト2のリンカ・コマンド・ファイルを使うことを前提にしているため，new演算子による領域確保が失敗することはない．そのため，ここでは失敗した場合の対策は行っていない．領域確保を失敗した場合の対策については，次回以降に説明する．

注9：orderはconstデータ・メンバなので，メンバ初期設定以外では値を設定することができない．

▶ 直接形FIRフィルタ実現のためのクラス

クラス`FIR_Direct`は，メンバとしてコンストラクタとメンバ関数`Execute()`をもっています．

コンストラクタは，基底クラスのコンストラクタにフィルタの次数を渡すだけで，それ以外の処理は行っていません．

フィルタ処理の本体はメンバ関数`Execute()`が担当します．このメンバ関数は基底クラスの純粋仮想関数に対応するものです．このメンバ関数の中に二つの`for`ループがあります．一つめの`for`ループは，式(1)の積和の計算に相当するものです．二つめの`for`ループは，遅延器のデータの移動に相当します．実行結果は戻り値として呼び出し側へ返されます．

▶ 転置形FIRフィルタ実現のためのクラス

クラス`FIR_Transpose`は`FIR_Direct`と同様に，メンバとしてコンストラクタとメンバ関数`Execute()`をもっており，コンストラクタは，基底クラスのコンストラクタにフィルタの次数を渡すだけで，それ以外の処理は行っていません．

リスト5　FIRフィルタ用のクラスを利用した低域通過フィルタ(FIR_LPF.cpp)

```
// -------------------------------------------------------
//      FIRフィルタ用クラスを使った低域通過フィルタ
//      (ポリモーフィズムの例)
//      標本化周波数 24.0 kHz
//      通過域端周波数     0.6 kHz
//      阻止域端周波数     0.9 kHz
//      次数  180
//      通過域での偏移     0.00315303 dB
//      阻止域での偏移     -50.02545113 dB
//      Parks-McClellan法による設計
// -------------------------------------------------------
#include "AIC23_Polling.hpp"
#include "FIR_class.cpp"

const int ORDER = 180;
const float hn[ORDER+1] = {
    -1.68359341e-03,   -2.08459665e-04,   -1.86972018e-04,
    -1.39776784e-04,   -6.59108248e-05,    3.47060843e-05,
     1.61105363e-04,    3.10761099e-04,

(省略)

    -6.59108248e-05,   -1.39776784e-04,   -1.86972018e-04,
    -2.08459665e-04,   -1.68359341e-03};

int main()
{
    float ch_in[2], ch_out[2];
    AIC23_Polling codec(codec.fs24kHz);


    BaseFIR *ptrFIR[2];
    ptrFIR[0] = new FIR_Direct(ORDER);
    ptrFIR[1] = new FIR_Transpose(ORDER);


    while(1)
    {
        codec.Read( ch_in) ;
        for (int m=0; m<2; m++)
            ch_out[m] = ptrFIR[m]->Execute(hn, ch_in[m]);
        codec.Write( ch_out) ;
    }
}
```

フィルタ処理の本体であるメンバ関数`Execute()`での処理内容は，`FIR_Direct`の場合とは異なっています．それは`for`ループが一つだけだということです．この`for`ループでは式(3)に相当する計算を行います．転置形FIRフィルタでは遅延器のデータの移動は必要ありません．実行結果は`FIR_Direct`の場合と同様に，戻り値として呼び出し側へ返されます．

● FIRフィルタのクラスを使ったプログラム

作成したFIRフィルタ用クラスを使ったプログラムの例をリスト5に示します．フィルタの係数は一部を省略しているので，すべての係数を知りたい場合には，InterGiga No.33に収録するファイルを参照してください．

フィルタの係数は参考文献(3)に付属するFIRフィルタ設計プログラムを利用しました．設計の際に与えたパラメータを表1に示します．また，設計されたフィルタの周波数特性を図3に示します．

メイン・プログラムでは，`AIC23_Polling`クラスのオブジェクト`codec`を宣言しています．ここでは標本化周波数をデフォルトの値である48kHzではなく24kHzに変更するため，それに対する引き数も指定しています．

このときのシンボル`fs24kHz`は`AIC23_Polling`クラスの基底クラス`AIC23_Base`で定義されています．したがって，リストの例のように`fs24kHz`の前に`codec.`を付ける必要があ

表1　FIRフィルタの設計時に与えたパラメータ

次数	180	
標本化周波数(kHz)	24	
	帯域1(通過域)	帯域2(阻止域)
下側帯域端周波数(kHz)	0.0	0.9
上側帯域端周波数(kHz)	0.6	2.0
利得	1	0
重み	1	5

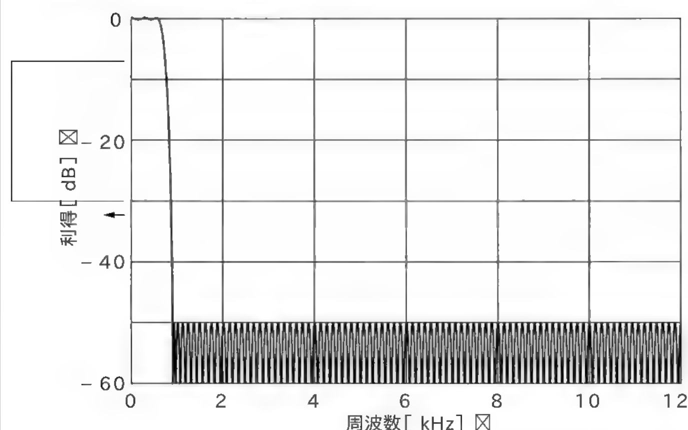


図3　作成するFIRフィルタの周波数特性

注10：この他の記述方法もある．その例については前回(2004年3月号)掲載のリスト3を参照のこと．

ります[注10]．

次に，BaseFIRクラスのポインタ配列ptrFIRの宣言を行います．このポインタに，new演算子で生成したFIR_DirectクラスおよびFIR_Transposeクラスのオブジェクトのポインタを代入します．ところで，C++は型を厳密に扱うので，基本的には異なるクラスのポインタの代入は許されません．そのため，この代入はコンパイル・エラーが出そうですが，この場合はだいじょうぶです．C++では基底クラスのポインタへ派生クラスのポインタを代入することは言語仕様として許されているからです．

whileブロックではアナログ信号の入力，左右チャネルのフィルタの実行，アナログ信号の出力を行います．このときのフィルタは，左チャネルを直接形で，右チャネルを転置形で実現するようになっています．

このwhileブロックの中で，ポインタptrFIRはBaseFIRクラスのポインタになっているので，コンパイル時には，Execute()がFIR_TransposeクラスまたはFIR_Directクラスのいずれのメンバ関数かはわかりません．実行時になって初めて，どちらのクラスのメンバ関数であるかがわかります．このように，同じ名前の関数であっても，実際にはオブジェクトに応じて別の動作をするしくみがポリモーフィズムです．

ところで，new演算子で新しいオブジェクトを生成した場合には，そのプログラムが終了する前に，かならずdelete演算子でそのオブジェクトを破棄しなければなりません．ところが，このプログラムではwhileブロックが無限ループになっているため，オブジェクトの破棄は行っていないことに注意してください．本来であれば，whileブロックに中に，ループを抜け出す機構を記述し，ループを抜けたところでオブジェクトの破棄を行うようなプログラムにすべきです[注11]．しかし，ここではできるだけプログラムを簡潔にするため，そのようなことは行っていません．

なお，このプログラムではFIR_DirectおよびFIR_Transposeをnew演算子により動的にインスタンス化していますが，静的にインスタンス化することもできます．その場合は，リスト5の囲んである部分を次のように変更します．

＜リスト5の囲んである部分の変更＞

```
FIR_Direct LPF_D(ORDER);
FIR_Transpose LPF_T(ORDER);
BaseFIR *ptrFIR[2] = { &LPF_D, &LPF_T };
```

＊　　＊

次回は複素数クラスと，それを応用したFFTクラスを作成します．FFTクラスは，次回以降でフィルタを実現する際に使います．

注11：仮に，基底クラスBase FIRのオブジェクトをdeleteにより破棄する場合，Base FIRのデストラクタを仮想関数にすることの効果が現れてくる．

参考文献

(1) TMS320C6000 Chip Support Library API User's Guide, Texas Instruments，文献番号SPRU401F，2003年2月.
(2) マーシャル クラインほか著，金澤 紀子訳；C++FAQ第2版　C++プログラミングを極めるためのQ＆A集，ピアソン・エデュケーション，2000年.
(3) 三上 直樹；C言語によるディジタル信号処理入門，CQ出版社，2002年.

みかみ・なおき　職業能力開発総合大学校 情報工学科

シニアエンジニアの技術草子 参拾七之段

◆オリンピック精神

旭 征佑

● オーバ・クロックの魅力

本誌の読者でオーバ・クロックということばを知らない人はいないだろう．一般的には，コンピュータのCPUを規格以上のクロックで動かし，より高いパフォーマンスを目ざすことをいう．そんなオーバ・クロックは，一昨年ぐらいまでは，どのパソコン雑誌の表紙も飾っていた代表的なキーワードで，パソコンの自作に取り組んだことのある人なら，だれでも一度は経験したことがあるテクニックだったと思う．

毎日の仕事にいそしんでいるサラリーマンだったら，オーバ・クロックは最高の趣味だったろう．なぜなら，仕事場では，組織の枠や不足がちの資源に縛られながらも，確実な結果を出すことが要求される．これに対し，お金や時間にこだわらず，自宅の趣味の分野でつねに最高を目ざす挑戦ができるのがオーバ・クロックだったからだ．

究極のオーバ・クロックを目ざす人も少なからずいた．人は彼らを畏敬（いふ）の念をもってオーバ・クロッカとも呼んだ．フライング販売される最高速のCPUを秋葉原で手に入れ，前人未到の高いクロックに挑戦する．あるいは廉価のCPUをアッと驚く高速で動かす．そのために，多くの資金をつぎ込んだ．特注の巨大な銅製のヒート・シンクに強大なファンをつけたり，特注グリスや，ペルチェ素子，水冷などを使用して排熱，冷却を図り，ベンチマークに挑戦して世界一速いマシンを作る．そのときの表情にはきっと鬼気迫るものがあったと思う．

オーバ・クロック・ブームは留まるところを知らなかった．一般のPC雑誌の特集だけでなく，パソコンの自作ブームともあいまって一般消費者までも大きな渦に巻き込み，文字どおり火がついた格好だった．何の知識もなくトライすると大きなトラブルを起こす可能性がある．これが，パソコンの自作を始めたばかりのユーザのプライドをくすぐったことは事実だろう．当然ながらCPUをだめにすることもあったようだ．しかし，そのリスクを背負っても，次々と挑戦するに足る醍醐（だいご）味があったのだろう．

● 昔からあったオーバ・クロック

クロックを高速化することで速く動かすというのは，CPUが発表されるずっと前からあるごく一般的な手法だった．CPUという高度に集積化されたチップでこの手法が試みられるようになったのは，たぶん1980年ごろのことだったと記憶している．その当時，もっともポピュラなCPUはZ80だった．もちろん，知らない人はいないだろう．ベスト・セラーCPUで，日本でもシャープやNEC，東芝から日立，川鉄にいたるまで多くのメーカがセカンド・ソース品を販売していた．現在でも産業機械などで多く使用されている．このCPUには，当時クロックが2.5MHzのZ80と4MHzのZ80Aがあったが，長く最前線で現役だったこともあり，高クロックのニーズが高まっていた．後年には6MHzのZ80B，8MHzのZ80Hが出たが，高価格だったことや，時代が16ビットへ移行しはじめたこともあり，あまり普及しなかった．

さて，このZ80だが，2.5MHzといいながら，実は4MHzや5MHzの高速のクロックでも動くものがあった．

少し説明しよう．半導体は製造工程でばらつきが出るのが当然だ．だからメーカがZ80を作る工程で，高クロックに耐えられるCPUをピックアップし，Z80Aとして出荷するというのだ．この話の信憑性は定かではないが，たぶん事実だろう．筆者の所持していた某社製のZ80Aは，Aの文字のみが，後からとって付けたようにずれて白く印刷されているのがはっきりわかったからだ．

さて，それがわかると，Z80Aを買うよりは，Z80をいくつか買い，高クロックで動かすテストをしてみたくなった．そして，5MHzで動いたときの喜びは飛び上がるほどだった．倍速で動かせるわけなので，これには大きな価値があった．何本も購入したので，最初からZ80Bなどを購入したほうが安かったかもしれないが，そこはまぁ，当たった喜びで十分に回収できたから，それで良かったのだ．

そうこうして，一部の自作マニアの間ではクロック・アップするのが，なかば当然のことだったという記憶がある．クロックを発生させるための各種の水晶も，いろいろな周波数のものを多数もちあわせていた．水晶だけ交換すれば，クロックを変更することができたからだ．

しかし，8086以降，次々に発表されたCPUは，上位互換が災い（？）し，クロック・アップするよりCPUを交換（換装）したほうが良くなっていった．そのためクロック・アップは話題にならなくなり，Pentium II 266やCeleron300Aのクロック・



アップ・ブームまでは影を潜めてしまう．ただし，CPU換装だったとしても，少しでも速く動かそうという欲求は，脈々と引き継がれたのは事実だ．

● 技術屋は何を目ざす

ところが，クロックが1GHzを超えたあたりから，オーバ・クロック熱がひんやりと冷めてきた．代わって静かなブームとなってきたのは静音化だ．確かに，今までのパソコンはうるさかった．静かになれば，夜中でもインターネットを楽しめる．

ある人はこういったとか．「車は速く走るだけが能ではない，ゆっくりと景色を見ながら，静かな車内会話を楽しみ，時間内にちゃんと目的地に着けばいい．ただ速いだけのオーバ・クロックはもう卒業だ」．こんな話を聞くと筆者は少しさみしくなる．

静音化そのものは，技術的・専門的だという人がいるかもしれない．たとえば，内部のファンの電圧を下げて回転数をコントロールする．流体軸受けのハードディスクを防音シートや防振ゴムでくるむ．流体力学を駆使して風の流れを計算し，ダクトを作成する．ただ，静音化を施すと，排熱が不十分になり，パソコンに致命的な打撃を与える可能性もある．ぎりぎりのところを追求するのは，オーバ・クロック同様，技術者としての醍醐味が味わえると思うかもしれない．

車でいえば，たとえばマフラで防音を施して，たとえば防音ウィンドウ・ガラスに交換，内部に防音シートを貼り巡らす．シートにも振動を減らすものを採用．さらにはエンジンのコンピュータを改造して回転数リミッタを付け…．お金はかかるが，確かに静かになりそうだ．しかし，何か足りないものがないだろうか．

なぜだろうか．筆者にいわせると，静音化というのは，技術を追求しているのではなく，高級化を追求しているように思えてしかたがないからだ．最近のCPUには，あり余るパワーを持ち，昔の電球よりも電気を食うものがある．誤解を恐れずに言えば，そんなものを使用して静音化してネット・サーフィンでもしていようものなら，燃費の悪い大排気量の外車を余裕の低回転で走らせ，スーパ・マーケットに買い物に行っているようにも思えてくる．まるでバブルではないか．

オーバ・クロックは，メーカ保証外の限界を超えて動かすので危険がともなう．高速ファンがうなりをあげてうるさいし，生暖かい風がよぎって気分を害するときもある．CPUだけ高速化しても周辺機器がついてこない場合があり，期待どおりの結果が出ないこともある．第一，そこまで速くする必要性も減ってきた．今までは，肥大化するソフトウェアにハードウェアが追いつかなかったのだが，最近は見直しが進み，ソフトウェアもリストラクチャされてきているからだ．



しかし，最高のパワーが出せるようエンジンをギンギンにチューニングし，音がうるさかろうと余計なものはすべて外して軽量化を図る．技術屋には，このほうがずっとおもしろいはずだ．ストレス解消になるし，車とは違い，人には迷惑をかけない．技術者である以上，つねに最上を目指すという挑戦はどこかで続けていくべきだ．挑戦を続けなくなったら終わりではないか．

古代ギリシャ以来，「より速く，より高く，より強く」は，数千年にわたって脈々と受け継がれてきた．人類の永遠の夢でもあり，これこそ本当のオリンピック精神だと筆者は思う．夢に向かって再び走りだす日が，必ずまた来ると筆者は思っている．

あさひ・しょうすけ　テクニカル・ライタ
イラスト　森 祐子

Engineering Life in

フリー・エンジニアという仕事(第一部)

▌今回のゲストのプロフィール
ボブ・アイゼンスタッド(Bob Eisenstadt): LSI設計エンジニアとして20年近くの経験を有する. VLSI Technology Inc.(現在はPhilipsの一部)でASIC設計の経験を積んだ後, Supermac, Radius, Silicon Graphics, 3DFX, Silicon Imageなどのグラフィックス関係の企業でLSI設計の外部スペシャリストとして活躍する. そのほかにスタートアップの設立の経験や, パテント取得の経験を有する. オフには運動, 造園, 家族と過ごす. マサチューセッツ州ボストン出身.

☆ レイオフを新卒で経験する

トニー▷ たいへんお久しぶりです! ボブさんとは私がまだ新卒のひよっこのころにお会いした同期ですよね…. さて, まずはシリコンバレーに来たきっかけから話してください.

ボブ☒ 私はボストン生まれなのですが, 父がMIT出身のエンジニアで, 兄もフロリダ大学の工学部の教授と, エンジニア一家でした. 私がコーネル大学を卒業するころに両親がカリフォルニアに引っ越したので, 私も西海岸で就職しようと思い, 卒業後シリコン・バレーに来ました. ディジタルとアナログの両方の半導体の勉強をしていたので, Signetics(現在Philipsの一部)に入り, プロダクト・エンジニアというプロセス・ラインのめんどうをみる仕事をしました.

トニー▷ あ〜. 私もインターンのときにプロダクト・エンジニアをやりました. 半導体の量産までのめんどうをみるので, 設計〜プロセス〜テストそしてマーケティングなどのグループと少しずつ接点があり, 良くいうと全体の流れがわかる仕事ですが, 悪くいうと使い走りみたいな仕事が多く, エンジニアリング的には中途半端な仕事になりますね.

ボブ☒ おっしゃるとおり, 使い走りが多いですよね. このときは結局, 新卒で入社してまもなく80年代半ばの半導体業界の不況に遭遇し, レイオフされてしまいました.

トニー▷ 私も84年に卒業予定だったのですが, 不況でなかなか新卒を取らない状態が続いていたことを覚えています. インターンを8か月ぐらいやって, それでまた単位も足りなかったので85年の卒業になりましたが, 当時はたいへんでした.

ボブ☒ 日本の半導体メーカが力を付けてきた時期で, アメリカのメーカがメモリなどから撤退を始める時期でした.

個人的には, そのころからですね. 自分の仕事は自分で責任を持たなければ…という意識が深く根付いたのは. 具体的にいうと, 会社がいうことをすべて信じないで, 自分の会社の業績などを外部からのレポートを見てチェックすることです. これは上場している会社だと, さまざまなアナリストの報告書を比較的容易に入手できるので, あとは業界紙の評価や, 同じような会社に勤めている友人を作るとか…ありとあらゆる方法を検討しなければ, と感じました.

また, しっかりと貯金をして, レイオフされても数か月は職探しができるようにするとか, 会社以外の人たちとネットワークを築くといった点です. とにかく, 自分をリニューアルして, 新しいチャンスにうまくめぐり合えるようにすることを心がけるようになりました.

そう思うと最近の若いエンジニア達はたいへんでしょうね. シリコン・バレーの物価も高いし, 遊ぶ場所とか増えましたし…. 以前は物価はそこそこだったし, あまり遊ぶ場所もなかったからお金のむだ遣いなどはしなかったと覚えています.

トニー▷ 今回の不況は現在も続き, 多くの若手エンジニア達がレイオフされたようですが, ボブさんのおっしゃるようなスキルを早く身に付けるのは大切ですよね. インターネット・バブルの時期に良い待遇で社会へ出ると, 仕事面でもプライベート面でも錯覚してしまいますからね.

ボブ☒ そうですね, 業績が良いと自分のスキルだと勘違いしてしまう人が多いですよね.「俺はエンジニアとして凄いんだ!」みたいに…. たまたま会社の業績が良かったとかいうこともありますから. また, 自分以外にも賢い人はたくさんいるわけだし, 会社の業績もサイクルがあるので良い時期がずっと続くという保証もなく, レイオフが当たり前であるぐらいの覚悟が必要だと思います. 私の場合は, 大学院に戻ることで電子工学の修士号を取り, これでASICの設計エンジニアとしてリニューアルを果たしました.

☆ ASICエンジニアにリニューアルする

トニー▷ 大学に戻ったり夜間で修士号や博士号を取るのは, エンジニアとしてのリニューアルをはかる王道ですよね. ここでボブさんと私はVLSI Technologyでめぐりあったわけです.

ボブ☒ そうです. VLSIにいるときも会社の制度を利用して夜間のMBAに通い, MBAを取得しました.

トニー▷ う〜ん, なかなか抜かりがないですね. 今からVLSI Technologyを振り返ると, ASIC設計の基本を学ぶには非常に良い環境でしたよね? 当時, 私は新卒のひよっこで, 何もかもが新しくて勉強, 勉強の日々でした.

ボブ☒ ASICということばがはやり始めたころで, プロセスから設計ツール, そしてIPまで幅広くやっていた会社だったので, ASICを学ぶにはとても良い環境でした. ただ手を広げすぎたのが後々落ち目になった原因だと思います.

また, さまざまな会社の設計を請け負ったり手伝ったりするので, その当時のメジャーなシリコン・バレーの会社と付き合うことができたのがのちの財産となっています. Apple, NeXT, Rolm, Unisys, Silicon Graphics, Radius….

トニー▷ あ〜, 私もAppleやNeXTに住み込んで設計の手伝いを行った経験があります. その後は?

ボブ☒ 当時のVLSIのメジャーなお客にAppleがあり, 付き合った多くのエンジニアがSupermacやRadiusに流れていた

ので，それの誘いを受けて2Dグラフィックスの世界に入りました．まあ，することは同じでASICの設計です．フロントエンドのロジック設計からレイアウト設計まで，一通りの流れをこなしました．

その後，当時はマックのモニタで有名だったRadiusがSupermacといっしょになったのですが，新会社はマックのクローンに手を出して，それが原因で会社の業績が一気に悪化しました．レイオフの可能性がプレス発表されていたので，そろそろ危ないと思い，知人に連絡を取り，そのまま休暇をとって家族とヨーロッパへ遊びに行ってしまいました．休暇後に自分の仕事が残っているかの保証なしでね（笑）．休暇から戻って来たときにはドキドキしましたよ．まず会社に夜行って，カード・キーが使えるか試してみたり，机に最後の給与明細の通知が置いてないかチェックしました．その後，レイオフが実施されたのですが，その前に仕事が見つかったので会社を辞めていました．

☆ フリーエンジニアとして働く

トニー 就職されたのですか？

ボブ VLSIに残っていた知人が，Silicon Graphicsのお客に私が辞めることを知らせていました．営業系の知人だったのですが，ちょうどそのころ，Silicon Graphicsには大きなプロジェクトがいくつかあったので，猫の手も借りたい状態で，Silicon Graphicsの担当者は「ボブっていうエンジニアが辞めるので使ってみたら…」と声をかけてくれたそうです．それで，休暇後に担当者と会って，プロジェクトごとに働くコントラクタ[注1]になりました．まあ，このころにはもう就職して，エンジニアを続けるにもあまり仕事の保証がないのでフリーでも良いかなと思いつつスタートしました．おもしろいもので，一つ目のプロジェクトが終わるころにはもう一つがあり，結局三つのプロジェクトに参加したので長い間続きました．

トニー 90年代初期ですよね．ワークステーションがまだまだ花形でSilicon Graphicsが全盛期のころですね？

ボブ そうですね，コスト意識もあまりなかったし，とにかく最強の3Dグラフィックスを！といった感じでした．賢い人がたくさんいたおもしろい会社だったので，その後はとても残念でした．パソコンが主流になるというのを読み違えてそれからだめになりました．エンジニアの多くは，そのころからスタートしたほかの3Dグラフィックスの会社に行ってます．ATIや3DFXとか….

注1：日本流にいえば契約社員の意味．しかし，個人のフリー・エンジニアが直接会社と契約するケースがほとんどで，派遣会社に登録するなどの習慣はシリコン・バレーでは比較的少なく，フリーの専門エンジニアという意味合いが強い．



ボブ・アイゼンスタッド氏

トニー 当時のSilicon GraphicsやSunなどはデバイスの開発もたくさんやっていたし，ツールやソフトウェアの発注も多かったので，それでシリコンバレーが潤うし，また斬新な新しいことをやっていたので，それに育てられたほかの会社も多いですよね．ハードウェア記述言語も早い時期から使っていたし，Synopsysの論理合成ツールも早い時期から導入していましたね．

ボブ いつの時期もシリコン・バレーを代表する大きな会社があり，そこが新しい技術を利用するという例が多いですよね．最近だとCiscoとかNVIDIAとかでしょうか？　その後は，Silicon Graphicsの仕事が終わってから長期の休暇を取り，家族サービスをしたり家の手入れや庭に果物の木を植えて数か月間を過ごしてました．

その次には，3DFXに行ったSilicon Graphicsのエンジニアから声がかかり，そこでまたコントラクタとして働くことになりました．3DFXも，当時は3Dグラフィックスでは注目された会社ですが，ボード会社を買収することになり，これが命取りとなって業績が悪化してしまいました．ただ，他社と違って，非常に急なレイオフでした．ある金曜日に全員が呼ばれて，社長がみんなレイオフです…といった感じで宣告したのです．普通なら徐々にスタッフを減らしていき，徐々に落ちていくのが普通で，周りの社員達も薄々気が付くのですが，この場合は違いました．私は社員ではなかったので，まったく気になりませんでしたが，社員の方達はたいへんだったと思います．

次回の予告

引き続きフリー・エンジニアとしての生活などについてうかがう．また，今後のエンジニアリングについての話題が出る．

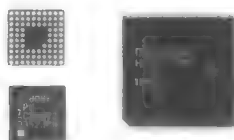
トニー・チン　htchin@attglobal.net　WinHawk Consulting

●16ビット・1チップ・マイコン

H8/38086F

- CPUコアは，16ビット「H8/300H」を搭載.
- 高精度なデルタ・シグマ型14ビットA-D変換器を搭載.
- A-D変換器は，フルスケール・レンジが3.0Vでの非直線性誤差を，従来と比較して約30倍改善している.
- PGAや基準電圧用のバンド・ギャップ・リファレンス回路などの周辺機器を内蔵.
- 1.8Vでの低電圧動作が可能で，電池駆動機器におけるバッテリ寿命の長時間化を図ることができる.
- 8種類の低消費モードを搭載しているため，機器の動作に合わせた最適なモードを選択して低消費電力化を図ることができる.

●価格：¥800～¥900（1,000個時）



■（株）ルネサス テクノロジ
TEL：03-5201-5224

●マルチメディア・アプリケーション・プロセッサ

STn8800

- 0.13μmのプロセス技術により，300mm（12インチ）のウェハで製造される.
- 携帯電話やPDAなどの携帯端末上での音楽再生や写真撮影，さらに端末間でのビジュアル通信をリアルタイムに実行可能.
- スマート・アクセラレータを基礎としたアーキテクチャを採用することで，電力消費とコストを抑えながら，クラス最高のビデオ品質を提供.
- CMOSイメージ・センサ，ワイヤレス接続，スマート・カードなど，さまざまな周辺機器に対応するインターフェースを幅広く搭載しており，端末機器の「マルチメディア・ハブ」として使用することで設計の簡素化が図れる.

●価格：下記へ問い合わせ



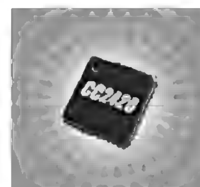
■STマイクロエレクトロニクス
TEL：03-5783-8361 FAX：03-5783-8216

●RFトランシーバIC

CC2420

- ノルウェーのChipconAS社が開発したRFトランシーバLSI.
- 独自のSmartRF03テクノロジをベースに，CMOS 0.18μmプロセスで開発.
- 2.4GHzでIEEE802.15.4準拠，Zigbeeに対応している.
- EN300 440およびFCC CFR47 Part15，ARIB-STD-T66の標準規格に準拠し，2400MHz～2483.5MHzの帯域をカバーし，最小動作電圧1.6Vで動作.
- パケット処理，データ・バッファリング，バースト伝送，アドレス認識，クリア・チャネル評価，リンク品質表示およびタイミング情報の機能をサポート.

●価格：¥500（5,000個時）



■テクセル（株）
TEL：03-5467-9273
E-mail：chipcon@teksel.co.jp

●デジカメ向け画像処理用LSI

Millennia-3/ Millennia-3MM

- 色処理部には独自の色処理アルゴリズムを用いており，エッジ強調機能やノイズ除去機能を大幅に改善.
- 画像処理回路を搭載しているため，大量の画像データの色処理やJPEGへの圧縮/伸張などを，低消費電力で高速に処理することが可能.
- Millennia-3MMは，MPEG-4のCODECを搭載．VGAサイズの動画を30フレーム/sで圧縮/伸張できるため，ディジタル・ビデオ・カメラなみの動画撮影が可能．動画内の物体の動きを予測する「動きベクトル予測機能」や，画像の高周波ノイズをカットする「プレフィルタ」をハードウェアとして搭載.
- パソコンなどへの高速なデータ転送を可能にするUSB2.0インターフェースを搭載.
- ディジタル・カメラ向けのディジタルLCDインターフェースを搭載.

●価格：¥4,000（Millennia-3）
¥5,000（Millennia-3MM）

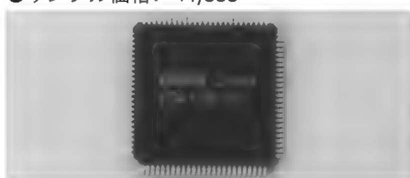
■富士通（株）
TEL：03-5322-3325
E-mail：edevice@fujitsu.com

●ディジタル・アンプ用コントロールLSI

YDA136

- ホーム・シアタ用のディジタル・アンプ・システムを構成するために必要な回路を集積したLSI.
- 24ビット・ディジタル・オーディオ信号とアナログ・オーディオ信号の入力インターフェース回路，2チャネルの電子ボリューム回路，2チャネルのディジタル・パルス変調回路，パワーMOSFET用プリンタ・ドライバ回路，パルス・フィードバック回路，マルチチャネル・データ・インターフェース回路を1チップ化.
- 内蔵のパワーMOSFET用プリンタ・ドライバに，パワーMOSFETを接続することで，60W～100W/CH@6Ω，100W～150W/CH@4Ω出力のディジタル・パワー・アンプの構築が可能.

●サンプル価格：¥1,000



■ヤマハ（株）
TEL：0539-62-5444

●モバイルRAM

EDL6416CABH/ECL6416CACN EDL6416BABH/ECL6416BACN

- EDL6416CABH/ECL6416CACNは，Vdd/VddQが1.8Vでクロック周波数が100MHz，EDL6416BABH/ECL6416BACNは，2.5Vで133MHzの携帯電話，モバイル機器に適する64Mビット・モバイルRAM.
- 1.8V品は，低消費電力テクノロジにより，100μAのセルフ・リフレッシュ電流を実現.
- 128Mビットモバイル RAMと共通の8×8mmFBGAパッケージに封入しているために，将来的な容量拡大に対応.
- チップ周辺にボンディング・パッドを配置したエッジ・パッドを採用することで，MCP/SIPへの実装が容易.
- 周囲温度によって自動的にセルフ・リフレッシュ周期を変更し，消費電流を抑えるAutoTCSR機能を搭載.

●価格：下記へ問い合わせ

■エルピーダメモリ（株）
TEL：03-3281-1648

●STB対応チップ・セット

STi5528/STi4629

- デュアルTVとデュアルPVRの機能をサポートしており，二つの32ビットRISCプロセッサを搭載している．
- 二つのビデオ・データとオーディオ・データを個別にデコードすることができるため，2台のテレビを単一のSTB(セット・トップ・ボックス)に接続することが可能．
- 二つのオーディオ・データとビデオ・データをデコードしながら，二つのチャネルを一つのハードディスクに録画できるデュアルPVR機能を提供．
- STi5528は166MHzで動作するST20コアを採用し，最大四つのフロント・エンドとATA-6準拠ハードディスク・ドライブ・インターフェースから，七つのトランスポート・ストリームのデマルチプレックスおよびデスクランブルを行うことができる．
- 衛星，有線または地上波のSTBで使用可能．

●価格：下記へ問い合わせ

■STマイクロエレクトロニクス(株)
TEL：03-5783-8340 FAX：03-5783-8216

●携帯電話用カメラ・チップ・セット

VS6552/STV0974E

- CMOSイメージ・センサ・モジュール「VS6552」と画像処理用DSP「STV0974E」で構成される，携帯電話用カメラ向けのチップ・セット．
- VS6552は，同社のHCMOS7iプロセスで生産され，1ルクスの光量でVGA画像が得られる．
- 最小照度は1ルクス未満(F2.8レンズで S/N 0dB)，暗電流は25℃で30電子/sとなっている．10×8×6mmというコンパクト・サイズを実現．
- STV0974Eは，6×6mmのリード・フリー・パッケージである56TFBGAを採用し，センサ制御と信号調整，画像強調，画像クロップと縮尺，ビュー・ファインダ生成，JPEGファイル生成を含むイメージ・キャプチャなどの多彩な機能をもつ．
- MMSを見据えて設計され，ショート・ビデオ・クリップや限られたファイル・サイズのイメージ・キャプチャの実行が可能．

●サンプル価格：¥5,000

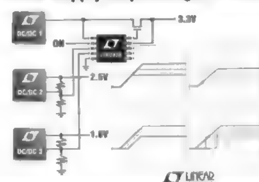
■STマイクロエレクトロニクス(株)
TEL：03-5783-8340 FAX：03-5783-8216

●電源トラッキング・コントローラ

LTC2923

- 帰還ノードに電流を流すことによって，電源の起動と切断を制御し，複数電源を容易に設定可能．
- 2本の抵抗を使用するだけで，電源をマスタ信号に応じてラインナップすることが可能．
- 第3の電源，充電コンデンサ，外部生成された信号のいずれかがマスタ信号として可能．
- 2本の抵抗を選択することにより，電源のランプ・アップとランプ・ダウンを同時に設定でき，固定電圧オフセット，遅延挿入，ランプ・レートのオプション設定が可能．
- 二つの電源出力が，パス素子の損失なしにトラッキングを行ったり，シーケンスに従うことができる．

●サンプル価格：¥330(1,000個時)



■リニアテクノロジー(株)
TEL：03-5226-7291 FAX：03-5226-0268

●広帯域電流帰還アンプ

OPA695

- 高ゲイン動作(G=+8)において，4300V/μsの高いスルー・レートと450MHzを越える帯域幅をもつ，低消費電力のIFアンプ・アプリケーション向けのアンプ．
- ビデオ信号のライン・ドライバ・アプリケーションで，一般的な低ゲイン動作(G=+2)において，1.4GHzの帯域幅と2500V/μsのスルー・レートを提供．
- 単一+5V～+12V電源で動作し，既存のIFアンプと比較して，より低消費電力でより高い出力IP(インターセプト・ポイント)特性を提供．
- 電源電流は，+25℃においてトリミングされ，12.9mAと低く設定されている．
- トリミングと広い温度範囲で無信号時消費電流の変動が小さいことにより，周囲温度の変動に対してもシステムの消費電流を低く抑えることが可能．
- 0℃～+70℃の商用温度範囲ならびに，−40℃～+85℃の工業温度範囲で規定される．
- オプションのディセーブル制御ピンにより，電源電流を170μAに抑えることが可能．

●推奨価格：¥195(1,000個時)

■日本テキサス・インスツルメンツ(株)
FAX：0120-81-0036

●パラレル・ポート・トランシーバ

74LVC161284TTR/74LVCZ161284ATTR

- 74LVC161284TTRは，IEEE1284に準拠する低電圧高速トランシーバ．
- 74LVCZ161284ATTRは，さらにエラー・フリーのパワー・アップ機能を搭載．
- 一つで，既存製品2個分の機能をサポート．
- PCと周辺機器間の双方向パラレル通信に使われているIEEE1284-Ⅰ，IEEE1284-Ⅱ規格に対応．
- トランスレーション機能を搭載しているため，5V信号でケーブル側からインターフェースへの出力が可能．
- 各オープン・ドレイン出力には，プルアップ抵抗が組み込まれている．

●サンプル価格：
74LVC161284TTR ¥95(1,000個時)
74LVCZ161284ATTR ¥105(1,000個時)



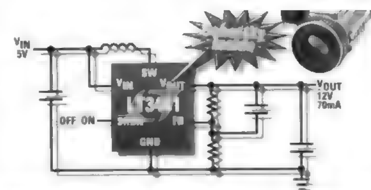
■STマイクロエレクトロニクス(株)
TEL：03-5783-8240 FAX：03-5783-8216

●DC-DCコンバータ

LT3461/LT3461A

- 40V，300mA(I_{SW})，低$V_{CE(sat)}$(260mV)のスイッチを内蔵．
- ショットキ・ダイオードの内蔵によって，外付けダイオードのサイズとコストを節減．
- ソフト・スタート機能の搭載で，高い突入電流を防止．
- 1.3MHz/3MHz 固定周波数動作の電流モード・アーキテクチャの採用で，低ノイズ動作が可能．
- 入力電圧範囲は，2.5～16V．
- 消費電流は動作時に2.8mA，シャット・ダウン時には1μA以下．
- TFT-LCDバイアスやxDSLなど，低ノイズと小さな実装面積を必要とする用途に適する．

●サンプル価格：¥185～(1,000個時)



■リニアテクノロジー(株)
TEL：03-5226-7291 FAX：03-5226-0268

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp (編集部)

●低オン抵抗パワーMOSFET

STD150NH02L

- ドレイン-ソース間電圧V_{DS}が24V，最大ドレイン電流I_Dが150AのNチャネルMOSFET.
- オン抵抗$R_{DS(on)}$は0.0035Ωと小さく，ゲート・チャージと熱抵抗を低減.
- $R_{DS(on)}$は標準で，10Vで0.003Ω，5Vで0.005Ωとなっており，部品における伝導損失を低減.
- 同社の第3世代StripFET製造テクノロジで設計されている.
- 0.6μmプロセスでは，独特のメタライゼーション技術およびボンドレス・アセンブリ技術を使用し，標準的なDPAKアウトライン・パッケージからハイ・パフォーマンスを実現.

●サンプル価格：¥400（10個時）



■STマイクロエレクトロニクス（株）
TEL：03-5783-8240 FAX：03-5783-8216

●車載用モータ・コントローラLSI

IR3220S

- ハイ・サイド側のHEXFETパワーMOSFET2個を内蔵.
- ロー・サイド側のMOSFET2個と受動部品10個程度を外付けすれば，フル・ブリッジのモータ・コントロール回路を構築できる.
- 順方向，逆方法，ブレーキ，ノンブレーキの各モードを備えているため，マイコンで制御する必要がない.
- 最大定格電圧は40Vで，パッケージは20ピンSOP.
- ハイ・サイド側とロー・サイド側の両方の回路に対して貫通，温度上昇（165℃），過電流（30A），過電圧クランプ（40V），低電圧ロック・アウト（UVLO）の各保護機能を装備.
- 発振周波数20kHzの内蔵PWM回路を使って，突入電流を制限するプログラマブルなソフト・スタート機能を装備し，外付けの抵抗とコンデンサの時定数で特性設定が可能.
- ヒートシンクなしで，最大6Aの出力電流が得られる.

●サンプル価格：¥750

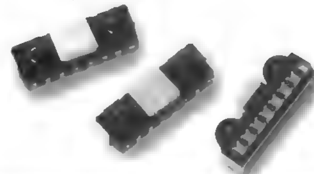
■インターナショナル レクティファイアー ジャパン（株）
TEL：03-3983-0086 FAX：03-3983-0642

●赤外線トランシーバ

HSDL-3220 赤外線トランシーバ

- IrDAのデータFIR規格低電力オプションに準拠した，携帯電話やPDAに搭載される赤外線データ通信モジュール.
- 8.0×3.0×2.5mmの低背パッケージに納め，小型サイズを実現.
- 通信速度は，9.6kbps～4.0Mbps.
- I/O V_{cc}機能を搭載しており，外付け部品を追加することなく，コントローラICからリファレンス電源をとることが可能.
- 動作温度範囲は，－25～+70℃.
- 電源電圧は，2.7V～3.6V.
- シャット・ダウン時電流は，0.1μA（代表値）.
- リンク長は，50cm（代表値）.
- 鉛フリー・パッケージを採用.
- LED常時点灯防止機能付き.

●サンプル価格：¥270



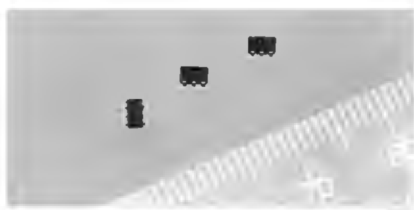
■アジレント・テクノロジー（株）
TEL：0120-61-1280

●CMOSボルテージ・レギュレータ

S-1200シリーズ

- 高リプル除去率でありながら，動作時18μA（typ.）の低消費電流を実現.
- 入力/出力コンデンサともに0.1μFの小型セラミック・コンデンサの使用が可能.
- 70dBのリプル除去率を実現しており，ノイズを極力抑制することができるため，携帯情報機器やディジタル・カメラなどに適する.
- ±1.0%の出力電圧精度は，製品の消費電流を減少させ，長時間使用を可能にする.
- SOT-23-5パッケージ（2.9×2.8×1.3mm）の小型パッケージを採用.

●サンプル価格：¥100



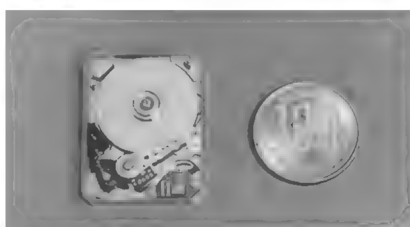
■セイコーインスツルメンツ（株）
TEL：043-211-1193
URL：http://www.sii-ic.com/

●ハード・ディスク・ドライブ

0.85型HDD

- 従来の1.8型HDDの1/4程度のサイズ（ディスク径：約2.2cm）で，モバイル機器への搭載に適する.
- 記憶容量は，2～4Gバイトを予定しており，音楽や映像などの大容量コンテンツを保存することが可能.
- 省スペース化，軽量化，低消費電力化，ユビキタス環境対応など，ディジタル家電環境のニーズに応える.

●価格：下記へ問い合わせ



■（株）東芝
TEL：03-3457-2100

●プラズマ・ディスプレイ

PX-61XM2P

- 独自のフル・ディジタル信号処理回路「Digital AccuDevice」を2系統搭載することにより，子画面のフルディジタル処理を実現.
- すべての入力系統どうしのサイド・バイ・サイド，ピクチャinピクチャによる2画面表示が可能で，15パターンの表示方法から選択が可能.
- サイド・バイ・サイドでは，画面に余白を作らないフル表示で，3パターンのアスペクト比を選択可能.
- サイド・バイ・サイドで，2画面をそれぞれ900%まで拡大できるため，細かいデータの拡大が可能.
- パソコン信号とビデオ信号をタイムラグなく切り替える「高速切り替え機能」を搭載.

●価格：下記へ問い合わせ



■NECプラズマディスプレイ（株）
TEL：03-3798-7273
URL：http://www.nec.co.jp/plasma/

●ディジタル・マルチメータ

VOAC7523/7522/7521

- 5.5桁機種の中では最高のカウント表示である，509999桁をサポート．
- デュアル表示を可能にしているため，1台で2台分の測定ができる．
- 完全に絶縁された，直流の測定入力を2チャネル設定することで，異なる回路の電圧比較を行うことができる．
- 標準インターフェースとしてRS-232-Cを搭載し，オプションでEthernet 10Base-T，GP-IBの選択が可能．
- タイムスタンプの分解能は10ms．

●価格： VOAC 7523 ¥145,000
VOAC 7522 ¥99,800
VOAC 7521 ¥89,800



■岩通計測（株）
TEL：03-5370-5474 FAX：03-5370-5492

●リアルタイム・スペクトラム・アナライザ

RSA3300A/RSA2200Aシリーズ

- 無線タグをはじめとしたRFIDやディジタル家電，高機能レーダなどの無線応用機器開発向けの製品．
- RF信号のシームレスな取り込み/保存や，過渡状態の取り込みが可能．
- 周波数軸解析，時間軸解析だけでなく，一つの画面の中でお互いの現象確認を時系列的にリンクして行えるため，バースト信号でも瞬時スプリアス観測が可能．
- RSA3300Aシリーズでは，レベル・トリガ機能に加えて周波数マスク・トリガを搭載．取り込みメモリも最長256Mバイトまで搭載でき，長時間の捕捉が可能．オプションで，ディジタル変調解析機能の搭載が可能．

●価格： RSA3300A ¥3,240,000～
RSA2200A ¥2,760,000～

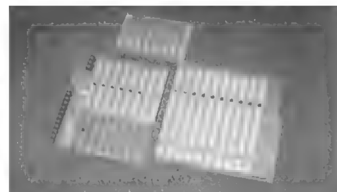
■日本テクトロニクス（株）
TEL：03-6714-3010 FAX：0120-046-011
URL：http://www.tektronix.co.jp/

●CompactPCIバス用バック・プレーン

CPCIシリーズ

- PICMG2.16R1.0，PICMG2.17R1.0規格に準拠したバック・プレーン，およびPICMG2.0R3.0規格に準拠したVAタイプ・バック・プレーン．
- ファブリック・ボードに，その他のボードが1対1で接続されているため，複数ボードの同時制御が可能．
- 二重化されたファブリック・ボードにより，安定したシステム構築が可能．
- CPCI-PSB-8S-R-Dでは，Ethernet規格をバック・プレーン上のP3コネクタに取り込んだ仕様．
- CPCI-SFB-10S-R-CDでは，LVDS規格をバック・プレーン上のP3コネクタに取り込んだ仕様．

●サンプル価格：¥24,000～¥130,000



■ケル（株）
TEL：042-374-5801 FAX：042-374-5887

●JTAG専用エミュレータ

EJ-Debug

- 100×86×23mmの小型サイズ．
- JTAGインターフェースに対応しており，JTAGポートを経由してターゲット・ボードのエミュレーション・デバッグが可能．
- パソコン上のコントロール・ソフトはソフィアシステムズ製デバッガ・ソフト「WATCHPOINT」を標準で添付．
- 本体ユニット上に「BATCH」ボタンを設けているので，ワンタッチで自動スクリプトを実行することが可能．
- パソコンからコマンドを操作することなく，バッチ処理が開始できるため，量産試作，量産品段階のターゲットの自動検査，バージョン・アップ装置，製造ラインでの使用やフィールドでのメンテナンスに適する．
- 対応CPUは，ARM7，ARM9，ARM7/9，SH-Mobile．

●予定価格：
¥298,000（ARM7，ARM9，SH-Mobile）
¥398,000（ARM7/9）

■（株）ソフィアシステムズ
TEL：044-989-7245 FAX：044-989-7005
E-mail：market@sophia-systems.co.jp

●無線LANカード

SL-5200

- 無線LANの暗号化認証方式WPAに対応しているため，RADIUSサーバを用いた認証セキュリティIEEE802.1x/EAPの環境下で利用可能．
- RADIUSサーバを利用しない，簡易的なWPA-PSKに対応．
- 暗号化セキュリティとして，米国政府の次世代標準暗号化方式OCB AES（128ビット）を搭載．
- 普及タイプのWEPには，従来の64/128ビットに加え，152ビットを追加サポート．
- 1枚で5.2GHz/54Mbps（IEEE802.11a準拠），2.4GHz/54Mbps（IEEE802.11g準拠），2.4GHz/11Mbps（IEEE802.11b準拠）の3モードの無線通信を実現．
- 通信モードの切り替えは，通信する無線LANアクセス・ポイントに合わせて自動的に行われる．
- アンテナ部分は，ダイバシティ方式の薄型設計．

●価格：オープン価格

■アイコム（株）
TEL：06-6792-4949

●PCMCIAビデオ・キャプチャ・カード

VCE-Pro VCE-B5A01

- VCE-Proは640×480ピクセル（30fps時），VCE-B5A01は640×480ピクセル（5fps時）で画像が取り込める．
- プラグ＆プレイCardBusに対応．
- リアルタイム表示，取り込み，ストアが行える．
- NTCS，PAL，SECAMからの信号を，シングル・フレーム，マルチ・フレームや標準AVIクリップスでキャプチャできる．
- プログラムによるシーケンスの取り込みや，外部トリガによるコントロールが行える．
- ノートPCを画像処理ツールにして，モバイル・イメージングの実行が可能．
- オプションのソフトウェア開発キットはDLLベースで，C++とVisualBasicをサポートしている．

●価格： VCE-Pro ¥75,000
VCE-B5A01 ¥45,000

■（株）アルゴ
TEL：06-6339-3366 FAX：06-6339-3365

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：（03）5395-2127，E-mail：mngnews@cqpub.co.jp （編集部）

●DVD装置用半導体レーザ

ML1XX23シリーズ

- 独自の端面窓構造やリッジ構造に加えて，キンク特性の改善を行うことにより，12～16倍速の記録型DVD装置に求められる，光出力200mWを実現.
- 75℃の高温動作を保証しており，光ディスク装置や光ピックアップ·モジュールの小型化などによる高密度実装に対しても，熱設計が容易になる.
- DVD± R/± RW，DVD-RAM装置など，すべての記録型DVD装置に適合.
- 中心波長は658nmで，発振いき値電流は65mA.
- 動作電流は，P_o=80mW，CWで150mA，P_o=200mW，パルスで270mA.

●価格：下記へ問い合わせ

■三菱電機（株）
TEL：03-3218-4772 FAX：03-3218-4862

●無線LANアクセス·ポイント

Dell TrueMobile 1170

- 無線LANカードを搭載したクライアント間や，インターネットおよび外部ネットワークへのアクセスを可能にする.
- 11Mbpsの高速なネットワーク接続を実現.
- アクセス·ポイント1台あたりに，約50クライアントの接続が可能.
- IEEE802.11bに準拠し，IEEE802.11gにも対応.
- WEPに準拠した64ビットおよび128ビットの暗号機能をサポートし，高度なセキュリティ機能をサポート.
- 3年間の保守サポートを提供.

●価格：¥69,000

■デル（株）
TEL：044-556-6190

●ネットワーク監視用サーバ機

TrueWitness1.5

- 不正アクセス，情報漏えいなどネットワークを監視するアプライアンス·サーバ.
- ネットワークを通過する99.9%以上のパケットを記録し，事後に解析を可能とする.
- 電子メールの送受信やホーム·ページの閲覧，掲示板への書き込みをパケット·レベルで記録，監視，解析が可能.
- データは自動的に分析，解析され，電子メールや添付ファイル，ヘッダ情報の復元閲覧が可能.
- タワー型とラックマウント型と用意.
- Webの利用状況，アクセス履歴をすべて保存可能.
- FTP利用を記録し，閲覧が可能.
- 侵入攻撃検知解析機能，利用者別ランキング表示機能などをサポート.

●価格：下記へ問い合わせ

■レーザーファイブ（株）
TEL：03-5818-6626 FAX：03-5818-6627

●基板設計シミュレーション·ソフトウェア

OPUSER XP

- ピン数やネットの制限なく，リアルタイム·アノテーション機能を搭載した基板設計シミュレータ.
- ダイレクトなアクセス·シミュレーションをサポート.
- EDSpiceシミュレーション·モデル作成機能付き.
- 日本語注釈や画面名などの入力が可能.
- 日本語文字の基盤配置やガーバ出力，ビットマップ入力が可能.
- ミックス·モード·シミュレーション·モデルの作成が可能.
- 独立した波形ビューワを内蔵し，ダイレクト·アクセス·オート·ルータを装備.
- 日本語のほか，8か国語の言語選択が可能.
- 日本製部品を含むANSI標準約35,000部品のライブラリを内蔵.
- DXF外形の入力が可能.

●価格：¥49,800～¥158,000（NC版）
¥298,000～¥498,000（Pro版）

■ユニクラフト（株）
TEL：03-3467-6041　FAX：03-3467-6042
E-mail：opuser-sales@unicraft.co.jp

●Webサービス構築ツール

LEIF/SourcePro C++ Edition 6

- 米国ローグウェーブソフトウェア社が開発した，フレームワーク「LEIF」およびC++ APIライブラリ「SourcePro」.
- LEIFは，Webサービスを提供するサーバとサービスを利用するクライアントを実装するためのフレーム·ワーク．XMLやWebサービス機能の組み込みを支援し，既存のC++アプリケーションをWebサービス対応アプリケーションに切り替え，J2EEや.NETで構築されたシステムとの融合を実現．WSDLからC++用のクライアント·プロキシとサーバ·スケルトンを自動生成.
- SourcePro C++ Edition 6は，システムの基盤となるデータベース処理，ネットワークやマルチスレッド対応など，汎用性の高い機能を持つC++ APIライブラリ．SourcePro Coreは，多目的な低レベルC++APIライブラリ．SourcePro DBは，異なるデータベースとの一元的な接続を実現．SourcePro NETは，インターネット対応アプリケーションを作成.

●価格：下記へ問い合わせ

■グレープシティ（株）
TEL：048-222-3001　FAX：048-222-1211
E-mail：rwsales@grapecity.com

●バーコード生成ツール

JBarCode 1.0J

- サーバ·サイドJava専用のバーコード·コンポーネントで，完全なJava APIライブラリ.
- CODE39，CODE93，CODE128，JAN8，EAN128，POSTNET，UPC/A，NW-7，カスタマ·バーコードなどの1次元バーコード，およびPDF417，QRコード，CODE49の2次元バーコード，合計17種類の規格に対応.
- バーコードの種類とデータを指定するだけの数行のプログラミングで，バーコードを実現.
- バーコード·イメージは，汎用性の高いJava Imageオブジェクトとして生成.
- サイズや色の変更，テキストの表示/非表示の切り替えやフォント指定，バーコードの回転など外見の調整が可能.
- 解像度の変更，チェック·ディジット自動付加，バーコードのサイズ指定やバー幅の指定により，高精度で信頼性の高いバーコード生成が可能.

●予定価格：¥58,000（1開発ライセンス）
¥100,000（1CPU運用ライセンス）

■グレープシティ（株）
TEL：048-222-3001　FAX：048-222-1211
E-mail：javasales@grapecity.com

●データ検索/集計ツール

軽技Web Ver5.0 for Oracle

- Webブラウザからデータベース情報の検索を可能にし，.NET Frameworkを採用することで高速処理を実現した検索ソフト．
- チューニングレスで，大規模データの検索が可能.
- 簡単な操作で，さまざまなデータ検索が可能.
- クライアントとしてはWebブラウザのみで利用可能で，そのほかのソフトウェアは不要.
- 検索条件の保存，共有，再利用，権限設定，URL公開などが可能.
- 表示方法を自由にカスタマイズ可能.
- 最大10テーブルを結合可能.
- 検索結果のXML，Excel転送により，自由なデータ加工をサポート．

●価格：¥500,000（1サーバ・ライセンス）



■富士通電機情報サービス（株）
TEL：03-5388-7825

●組み込み用統合化CASEツール

CasePlayer2 Ver2.2

- ソース・ファイルを登録するだけで，プログラム・ロジックを解析し，仕様書をはじめとした各種ドキュメントを自動生成.
- 組み込み向けC言語，各CPUアセンブラに対応.
- DSPほか，パラメータ・ファイル・サンプルを搭載.
- 自動生成された仕様書は，Word形式で出力.
- Cプログラム構文解析機能「Source Doctor」を装備.
- 自由な縮小倍率で，最適なレイアウト印刷が可能.
- 仕様書の一括HTML変換機能をサポート．
- 印刷，HTML変換時の仕様書カスタマイズ機能を搭載
- 目次印刷機能をサポート．
- フローチャート・ビューワ，モジュール構造図の画像ファイル出力が可能.
- 作成された仕様書を統合化する「仕様書ブラウザ」を搭載.

●価格：下記へ問い合わせ

■ガイオ・テクノロジー（株）
E-mail：case@gaio.co.jp
URL：http://www.gaio.co.jp/

●オブジェクト指向開発導入支援サービス

オブジェクト指向開発導入支援サービス

- オブジェクト指向開発を行う上で必要な知識を，短期集中で効率的に習得.
- オブジェクト指向入門では，クラスの見つけ方やUML，モデルを作成するなど，オブジェクト指向技術の基礎を学び，開発チームのメンバとコミュニケートするためのオブジェクト指向技術の基礎を習得.
- 統合開発環境であるEclipseを用いたJava開発で，Eclipseを使用する利点，利用方法，および演習によるプログラム，テスト，デバッグを習得.
- ラショナル認定コンサルタントを中心に，開発環境の作成，保守に対するアドバイス，分析や設計のモデリングに対するレビュー，実装に対するレビュー，開発プロセスに対するアドバイスと進捗レビューなどのサポートをオン・サイトまたはメールで行う．

●価格：

オブジェクト指向入門	¥750,000
Eclipseを用いたJava開発	¥600,000
基本アーキテクチャ構築	¥1,000,000
コンサルティング・サービス	¥2,000,000

■レッドフォックス（株）
TEL：03-5414-3315 FAX：03-5414-3316
E-mail：publicity@redfox.co.jp

●チャート描画ツール

RFFlow 5.0 日本語版

- 簡単な操作で描画やフローチャートの編集が可能.
- 1,600個を超える図形と編集機能を利用して，目的に合う図表にカスタマイズできる．
- 作成した図表をツール・ボックスに追加し，オリジナル・ツール・ボックスの作成が可能.
- Windowsアプリケーションに，作成したチャートを埋め込みまたはリンクで挿入することが可能.
- BMP，GIF，JPEG，HTML，WMF，EMF，ESP，PLT，PDFなどの画像ファイル形式でチャートを保存できる．
- WWWまたはそのほかのソースから画像をコピー＆ペーストすることが可能.
- 自動コネクタ，接続の保存，自動整列機能などの機能を装備.
- RFFlowがない場合でも，体験版（無料ダウンロード）でファイルの閲覧が可能.

●価格：¥27,000（ダウンロード版）
¥28,000（CD版）

■エクセルソフト（株）
TEL：03-5440-7875 FAX：03-5440-7876
E-mail：xlsoftkk@xlsoft.com

●PDF自動生成ツール

TallPDF.NET

- .NET環境で動作するアプリケーションに組み込むことで，動的に変化する内容をPDFに出力.
- ASP.NETやASP.NET Webサービスから，データベースの内容に応じた結果をPDF形式で出力することが可能.
- PDF作成用のクラス・ライブラリを，DOMに基づいて提供.
- プログラムからDOMを生成する代わりに，XML定義によるPDF作成が可能.
- 100%マネージ・コードで実装されており，一連の.NETクラスを通して機能を公開しているため，必要に応じて拡張が可能.
- イベント・ドリブン方式によりページ単位でドキュメントの生成/出力が可能なため，メモリ消費量の大幅な節約を実現.
- 40/128ビット・キー暗号化，ユーザおよびオーナ・パスワード，印刷許可，コピー許可，ユーザ権限などの設定が可能.

●価格：¥178,000（パッケージ版）
¥158,000（ダウンロード版）

■（株）エージーテック
TEL：03-3293-5283 FAX：03-3293-5270
E-mail：info@agtech.co.jp

●JSPカスタム・ライブラリ

GKitTaglib ExcelGenerator

- Jakarta POIライブラリを使用し，J2EEアプリケーション・サーバで，帳票や一覧表示などのExcelファイルを生成可能なライブラリ．
- 雛形のExcelのファイルを用意することで，必要な開発工数を大幅に削減.
- Web画面にExcelのファイルを配置，表示させることができる．
- Office97以上に対応.
- 動作環境OSは，Windows Server 2003，Solaris9（SPARC版），Red Hat Enterprise Linux Version3.
- 富士通 Interstage，日本IBM WebSphere，BEA WebLogic，Tomcatなどのアプリケーション・サーバに対応.

●価格：¥175,000

■（株）FFC
TEL：03-5324-1600　FAX：03-5324-1650
E-mail：market@ml.ffc.co.jp

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：（03）5395-2127，E-mail：mngnews@cqpub.co.jp　（編集部）



INFORMATION
海外・国内イベント/セミナー情報



海外イベント

3/8-11 **SAE 2004 World Congress**
COBO Conference/Exhibition Center, Detroit, MI, USA
SAE International
http://www.sae.org/congress/

3/15-19 **PCB Design Conferences West**
San Jose Convention Center, San Jose, CA, USA
UP Media Group
http://www.pcbwest.com/

3/16-18 **Wi-Fi Planet Conferences & Expo Canada 2004**
Sheraton Centre Hotel, Toronto, Canada
Jupitermedia
http://www.jupiterevents.com/wifi/canada04/

3/17-19 **SEMICON China 2004**
Shanghai New International Expo Center, Shanghai, China
SEMI
http://events.semi.org/semiconchina/V40/index.cvn

3/18-24 **CeBIT 2004**
Hannover Exhibition Center, Hannover, Germany
Deutsche Messe AG
http://www.cebit.de/
http://www.hannovermesse.co.jp/

3/22-24 **WIRELESS 2004**
Georgia World Congress Center, Atlanta, GA, USA
Cellular Telecommunications & Internet Association
http://www.wow-com.com/events/

国内イベント

3/2-5 IC CARD WORLD 2004／SECURITY SHOW 2004
東京国際展示場(東京ビッグサイト，東京都江東区)
日本経済新聞社
http://www.shopbiz.jp/pages/t_index.phtml?PID=0003&TCD=IC
http://www.shopbiz.jp/pages/t_index.phtml?PID=0003&TCD=SS

3/9-11 情報処理学会 第66回全国大会
慶應義塾大学 湘南藤沢キャンパス(神奈川県藤沢市)
(社)情報処理学会
http://www.ipsj.or.jp/10jigyo/taikai/66kai/

3/12 IPv6 Summit in KITAKYUSHU
北九州国際会議場大ホール
インターネット協会事務局
http://www.iajapan.org/ipv6/summit/index.html

3/17-18 第3回ケータイ国際フォーラム
京都府総合見本市会館 パルスプラザ(京都市伏見区)
ケータイ国際フォーラム推進会議
http://www.itbazaar-kyoto.com/forum/outline.html

3/17-19 nano tech 2004(国際ナノテクノロジー総合展・技術会議)
東京国際展示場(東京ビッグサイト，東京都江東区)
nano tech実行委員会
http://www.ics-inc.co.jp/nanotech/

3/19-21 フォトエキスポ2004
東京国際展示場(東京ビッグサイト，東京都江東区)
カメラ映像機器工業会／日本写真映像用品工業会
http://www.photoexpo2004.com/

4/7-9 IP FORUM 2004
東京国際展示場(東京ビッグサイト，東京都江東区)
(株)リックテレコム
http://www3.ric.co.jp/expo/ip2004/

4/7-9 EDEX2004(第19回 電子ディスプレイ展)
東京国際展示場(東京ビッグサイト，東京都江東区)
(社)電子情報技術産業協会
http://edex.jesa.or.jp/

セミナー情報

C言語によるはじめてのLinuxプログラミング
開催日時　：3月1日(月)～3月2日(火)
開催場所　：ディーアイエステクノサービス研修室(東京都文京区御茶ノ水)
受講料　：92,000円(税込，テキスト代を含む)
問い合わせ先：(株)エイチアイ ICP事業部，☎(03)3719-8155，FAX(03)5773-8661
http://icp.hicorp.co.jp/seminar/linux/clinux.asp

Linux開発環境構築
開催日時　：3月3日(水)
開催場所　：ディーアイエステクノサービス研修室(東京都文京区御茶ノ水)
受講料　：46,000円(税込，テキスト代を含む)
問い合わせ先：(株)エイチアイ ICP事業部，☎(03)3719-8155，FAX(03)5773-8661
http://icp.hicorp.co.jp/seminar/linux/linuxtool.asp

Linux Pスレッドプログラミング
開催日時　：3月4日(木)
開催場所　：ディーアイエステクノサービス研修室(東京都文京区御茶ノ水)
受講料　：49,000円(税込，テキスト代を含む)
問い合わせ先：(株)エイチアイ ICP事業部，☎(03)3719-8155，FAX(03)5773-8661
http://icp.hicorp.co.jp/seminar/linux/linuxtool.asp

シミュレータ テクニカルトレーニング
開催日時　：3月5日(金)
開催場所　：ガイオ・テクノロジー日本橋事業所セミナールーム(東京都中央区日本橋)
受講料　：無料
問い合わせ先：E-mail; seminar@gaio.co.jp
http://www.gaio.co.jp/event/regular_seminar.html

Linux GUIプログラミング
開催日時　：3月11日(木)
開催場所　：ディーアイエステクノサービス研修室(東京都文京区御茶ノ水)
受講料　：46,000円(税込，テキスト代を含む)
問い合わせ先：(株)エイチアイ ICP事業部，☎(03)3719-8155，FAX(03)5773-8661
http://icp.hicorp.co.jp/seminar/linux/clinuxgui.asp

プロトビルダー体験セミナー
開催日時　：3月11日(木)
開催場所　：ガイオ・テクノロジー日本橋事業所セミナールーム(東京都中央区日本橋)
受講料　：無料
問い合わせ先：E-mail; seminar@gaio.co.jp
http://www.gaio.co.jp/event/regular_seminar.html

パソコン実習によるRS232CからExcelへのデータ取り込み
開催日時　：3月18日(木)
開催場所　：CQ出版セミナールーム(東京都豊島区)
受講料　：20,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

自動化された論理検証ツールwinAMS評価コース
開催日時　：3月19日(金)
開催場所　：ガイオ・テクノロジー日本橋事業所セミナールーム(東京都中央区日本橋)
受講料　：無料
問い合わせ先：E-mail; seminar@gaio.co.jp
http://www.gaio.co.jp/event/regular_seminar.html

ディジタル画像処理入門
開催日時　：3月19日(金)
開催場所　：CQ出版セミナールーム(東京都豊島区巣鴨)
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

ビデオ信号の処理回路技術
開催日時　：3月20日(土)
開催場所　：CQ出版セミナールーム(東京都豊島区巣鴨)
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

リアルタイムOSの基礎
開催日時　：3月25日(木)
開催場所　：CQ出版セミナールーム(東京都豊島区巣鴨)
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

SH-Linuxマイコン入門
開催日時　：4月1日(木)
開催場所　：CQ出版セミナールーム(東京都豊島区巣鴨)
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

Linuxデバイスドライバ入門
開催日時　：4月2日(金)
開催場所　：CQ出版セミナールーム(東京都豊島区巣鴨)
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

開催日，イベント名，開催地，問い合わせ先の順
日程はすべて予定です．問い合わせ先にご確認のうえ，お出かけください．

読者の広場

Interface への声



2004年2月号特集「C++テンプレートプログラミングの世界」に関して

▷STLは以前から使っていたが，Boostは知りませんでした．なかなか良くできているようで，記事を参考にしながら試してみたいと思います．（JR9JUK）

▷2月号の別冊を見たとたん「おぉ黒いインターフェースだ！」と思わず声が出てしまいました．そして本文のほうも「おぉ，このフォントだったよなぁ」と喜んでしまいました．1988年からの読者です．あとはあのイラストがあったらねぇ～（satomum）

用字表記およびキャプション位置変更のお断り

「Interface」は2004年4月号より，CQ出版の統一誌面スタイルに合わせました．たとえば，用字用語で言えば，「バスプロトコル」→「バス・プロトコル」のように英単語ごとに「・（中黒）」を入れるようにしたり，図面や表のキャプションの位置を下にもってくる，といった仕様です．今後は，小社の各誌がいちだんと読みやすくなることと思います．

アンケートの結果

興味のあった記事（2004年2月号で実施，上位10位）

①第1章 テンプレートプログラミングの世界
②第4章 C言語で使えるコンテナライブラリ
③第2章 新世代テンプレートライブラリBoostの全貌
④初級ドライバ開発者のためのWindowsデバイスドライバ開発テクニック（第5回）
⑤フリーソフトウェア徹底活用講座 第14回）
⑥第3章 標準テンプレートライブラリSTLの概念，そして再考
⑦ハッカーの常識的見聞録(第37回)
⑧プログラミングの要(第9回)
⑨TOPPERSで学ぶRTOS技術（第4回）
⑩シニアエンジニアの技術草子（参拾五之段）

特集『C++テンプレートプログラミングの世界』についてのアンケートの結果

Q1　C++の「テンプレート」という言葉はご存知でしたか？
①はい（80%）
②いいえ（20%）

Q2　プログラミングをするうえで，テンプレートライブラリを使っていますか？
①いつも使っている（0%）
②たまに使っている（29%）
③まったく使っていない（71%）

Q3　現在おもに使っているプログラミング言語はなんですか？（複数回答可）
①C++（17%）　②C（34%）
③アセンブラ（9%）　④Java（3%）
⑤BASIC（17%）　⑥Perl（11%）
⑦その他（9%）

Q4　今後誌面で解説して欲しいライブラリがあれば，教えてください．
VCL/CLX，QT，.NET関連ほか

特集担当デスクから

☆組み込み機器にEthernetを搭載する場合，市販のネットワーク・インターフェース・コントローラ（NIC）を使うのが一般的でしょう．また最近では論理層部分をFPGAで実現するにしても，物理層は市販されているPHYデバイスを使うのがほとんどです．実用的にはFPGAだけで物理層から実現することはありません．

☆それでも今回，PHYデバイスすら使わないでEthernetコントローラを実現することにこだわったのは，ツイスト・ペア・ケーブル上を流れている生の信号をダイレクトに扱うことで，Ethernetの本来の姿を正しく理解できるようにするためです．

☆これまで，Ethernetの物理層レベルをさわってダイレクトにパケットのやりとりをしてみようといった場合でも，一般に市販されているNICの制御レジスタを直接制御してパケットをやりとりする程度でした．しかし，これではそのNICの使いかたを勉強しているだけで，果たして本当にEthernetの理解につながっているのだろうかという疑問が残ります．

☆もちろん，マンチェスタ符号のことなどまったく知らなくても，NICの使いかたを理解していればハードウェアの設計やドライバの作成はできます．また実際の設計開発の現場では，NICの使いかたを知っているほうが「使いものになる技術者」といわれるでしょう．

☆それでもせめて10Base-Tくらいは，生の波形を相手にできるくらいにEthernetを理解してほしいと考え，今回の特集を企画してみました．いかがだったでしょうか．

Interface

次号予告

2004年5月号は
3月25日発売です

フレッシャーズ特集
ようこそ組み込みシステムの世界へ！

組み込みシステムとは何か/組み込み向けOSとデスクトップOSの違い/クロス開発とは何か/組み込みシステムでのデバッグの実際/統合開発環境を用いた組み込み開発の事例

次号は新入社員を迎える4月直前ということで，フレッシャーズ向けに，「組み込みシステムとはなにか」について基礎的な解説を行います．

なじみ深いPCと組み込みシステムの世界では，違った概念や価値観が存在します．消費電力を抑えるためにx86以外のCPUを使ったり，メモリ消費を抑えるためにデスクトップ向け以外のOSを使ったりすることも多い組み込みシステム．それらに必要とされる技術は，ハードディスクのないシステムでのフラッシュ・メモリからのブートや，独自設計の基板へのOSのポーティング，クロス・コンパイルやシリアル・インターフェースを通じたデバッグなど，未知の世界が広がっています．

そこで次号では，これまで実践経験はないが，ハードウェア/ソフトウェアへの基本的な知識を持つ層を対象にし，組み込みシステムの世界について道しるべとなるような解説を行います．

編集後記

●大画面テレビの大半がこのままでは7年半後にはタダの箱，粗大ゴミになってしまうかもしれない．2011年7月24日を最後にアナログ波は停波することが電波法で決まっている．電子情報技術産業協会の統計によると，昨年11月のカラーテレビの国内出荷台数の87％はアナログ専用機だという．7年半後が心配だ．（檀）

●今月は特集と同時に増刊の編集作業も進行中で，えらい大変です（これを書いている時点も現在進行形！）．しかも特集は，ハードウェアの完成が遅れに遅れたため，送稿も遅れてしまい関係各所に多大なご迷惑を…．一時は，製作記事なのに物が未完成のまま掲載か!?と危ぶまれましたが，なんとか物も完成！　ホッ…（M）

●何となく元気が出ないときには，逆に思い切って体を動かすと効果的だということがわかった．それもバリバリにハードなほうが良い．一心不乱に動き回って，何も考えられなくなってきたとき，不思議な多幸感がえられる．人からは「中毒だ」といわれるのだが，これは一度体験してしまうと止められない．（＝IO）

●必要は学習の母（造語）．某方面に関して，漫然と「今後の参考になればいいや」と思って本を読んでいたのですが，理解度は6割止まり．それがいざ，必要ということになって死に物狂いで読み出したら理解出来ること出来ること．やっぱり人間，必要にならないと学習出来ないということでしょうか．（み）

●今ごろだが，昨日，1995年に登場したJR東日本常磐線E501系電車に乗った．この電車，発車と停車時にインバータの音が「ドレミファ～」（実際にはファソラシ～に近い？）に聞こえる独Siemens社製のVVVFインバータを採用．楽しいので一度乗ってみることをオススメ．でも毎日乗ったら1週間で飽きるかも．（もみ）

●いままで旅先でしかマッサージしたことがなかったのですが，肩こりと頭痛が続くので近所の整骨院に行ってきました．はじめは電気を通すだけでもドキドキでしたが今では気持ちいいし，丁寧にマッサージしてもらうと本当に極楽気分．毎日通ったほうが良いらしいけど，時間が取れないので週一の楽しみ．（Y2）

●恐らく今年は憲法改正のおおまかな道筋が示される戦後史の中で極めて大事な1年になるであろう．これから国民レベルまでさまざまな議論が展開されることを望みたい．しかしながら，一番大事なことは政治家が「戦闘地域，非戦闘地域」や「集団的自衛権」のような「解釈」という名の言葉遊びで「ごまかし」をしないことではないだろうか．（ちゃん）

●テレビのニュースで「オレオレ」詐欺が頻繁に報道されているが，本当にひっかかるものだろうか！？と思っていたら，最近私の周りで2人もおばあちゃんが騙されそうになりました．本人達は「本当に孫の声に聞こえた」と声を揃えて言っていました．1人は逮捕されました．（びび）

お知らせ

■ 読者の広場
本誌に関するご意見・ご希望などを，綴じ込みのハガキでお寄せください．読者の広場への掲載分には粗品を進呈いたします．なお，掲載に際しては表現の一部を変更させていただくことがありますので，あらかじめご了承ください．

■ 投稿歓迎
本誌に投稿をご希望の方は，連絡先（自宅/勤務先）を明記のうえ，テーマ，内容の概要をレポート用紙1～2枚にまとめて「Interface投稿係」までご送付ください．メールでお送りいただいても結構です（送り先はsupportinter@cqpub.co.jpまで）．追って採否をお知らせいたします．なお，採用分には小社規定の原稿料をお支払いいたします．

■ 本誌掲載記事についてのご注意
本誌掲載記事には著作権があり，示されている技術には工業所有権が確立されている場合があります．したがって，個人で利用される場合以外は，所有者の許諾が必要です．また，掲載された回路，技術，プログラムなどを利用して生じたトラブルについては，小社ならびに著作権者は責任を負いかねますので，ご了承ください．
本誌掲載記事をCQ出版（株）の承諾なしに，書籍，雑誌，Webといった媒体の形態を問わず，転載，複写することを禁じます．

■ コピー・サービスのご案内
本誌バック・ナンバの掲載記事については，在庫（原則として24か月分）のないものに限りコピー・サービスを行っています．コピー体裁は雑誌見開きの，複写機による白黒コピーです．なお，コピーの発送には多少時間がかかる場合があります．
- コピー料金（税込み）
  1ページにつき100円
- 発送手数料（判型に関わらず）
  1～10ページ：100円，11～30ページ：200円，31～50ページ：300円，51～100ページ：400円，101ページ以上：600円
- 送付金額の算出方法
  総ページ数×100円＋発送手数料
- 入金方法
  現金書留か郵便小為替による郵送
- 明記事項
  雑誌名，年月号，記事タイトル，開始ページ，総ページ数
- 宛て先
  〒170-8461　東京都豊島区巣鴨1-14-2
  CQ出版株式会社　コピー・サービス係
  （TEL：03-5395-4211，FAX：03-5395-1642）

■ お問い合わせ先のご案内
- 在庫，バック・ナンバ，年間購読送付先変更に関して
  販売部：03-5395-2141
- 広告に関して
  広告部：03-5395-2133
- 雑誌本文に関して
  編集部：03-5395-2122

記事内容に関するご質問は，返信用封筒を同封して編集部宛てに郵送してくださるようお願いいたします．筆者に回送してお答えいたします．

Interface
©CQ出版（株）　2004　振替　00100-7-10665
2004年4月号　第30巻　第4号　通巻第322号）
2004年4月1日発行（毎月1日発行）
定価は裏表紙に表示してあります

発行人／増田久喜
編集人／山形孝雄
編集／大野典宏　村上真紀　山口光樹　大竹友美　小林由美子
デザイン・DTP／クニメディア株式会社
表紙デザイン／株式会社プランニング・ロケッツ
本文イラスト／森　祐子
広告／澤辺　彰　中元正夫　菅原利江

発行所／CQ出版株式会社　〒170-8461　東京都豊島区巣鴨1-14-2
電話／編集部（03）5395-2122　FAX／（03）5395-2127
広告部（03）5395-2133　URL http://www.cqpub.co.jp/interface/
販売部（03）5395-2141　E-mail supportinter@cqpub.co.jp
CQ Publishing Co.,Ltd.／1-14-2 Sugamo, Toshima-ku, Tokyo 170-8461, Japan
印刷／クニメディア株式会社　美和印刷株式会社
製本／星野製本株式会社



日本ABC協会加盟誌
（新聞雑誌部数公査機構）　ISSN0387-9569